PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : H01J 13/00 | A2

(11) International Publication Number: WO 99/07007

(43) International Publication Date: 11 February 1999 (11.02.99)

(21) International Application Number: PCT/US98/15627

(22) International Filing Date: 28 July 1998 (28.07.98)

(30) Priority Data:
08/902,591 29 July 1997 (29.07.97) US

(71) Applicant: CATHARON PRODUCTIONS, INC. [US/US]; 2119 Route 66, Ghent, NY 12075-2408 (US).

(72) Inventors: FEINBERG, Michael, A.; 2119 Route 66, Ghent, NY 12075-2408 (US). FEINBERG, Matthew, A.; 2119 Route 66, Ghent, NY 12075-2408 (US).

(74) Agent: SUDOL, R., Neil; Coleman Sudol, LLP, 13th floor, 270 Madison Avenue, New York, NY 10016 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, HR, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: COMPUTERIZED SYSTEM AND ASSOCIATED METHOD FOR OPTIMALLY CONTROLLING STORAGE AND TRANSFER OF COMPUTER PROGRAMS ON A COMPUTER NETWORK

(57) Abstract

A computerized system and an associated method for optimally controlling storage and transfer of computer programs between computers on a network to facilitate interactive program usage. In accordance with the method, an applications program is stored in a nonvolatile memory of a first computer as a plurality of individual and independent machine-executable code modules. In response to a rquest from a second computer transmitted over a network link, the first computer retrieves a selected one of the machine-executable code modules and only that selected code module from the memory and transmits the selected code module over the network link to the second computer.

ENSDOCID: <WO___9907007A2_I_>

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

BNSDOCID: <WO 9907007A2 I >

COMPUTERIZED SYSTEM AND ASSOCIATED METHOD FOR OPTIMALLY CONTROLLING STORAGE AND TRANSFER OF COMPUTER PROGRAMS ON A COMPUTER NETWORK

Background of the Invention

This invention relates to computing and communications on a computer network. More specifically, this invention relates to a computerized system and an associated method for optimally controlling storage and transfer of computer programs between computers on a network to facilitate interactive program usage.

In the past several years, there has been an ever increasing number of individuals, corporations and other legal entities which have obtained access to the global network of computers known as the Internet. More precisely described, the Internet is a network of regional computer networks scattered throughout the world. On any given day, the Internet connects roughly 20^2 million users in over 50 countries. That number will continue to increase annually for the foreseeable future.

The Internet has provided a means for enabling individual access to enormous amounts of information in several forms including text, video, graphics and sound. This multi-media agglomeration or abstract space of information is generally referred to as the "World-Wide Web," which is technically a "wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents." To obtain access to the World-Wide Web, a computer operator uses software called a browser. The most popular browsers in the United States are Netscape Navigator and Microsoft's Internet Explorer.

The operation of the Web relies on so-called hypertext for enabling user interaction. Hypertext is basically the same as regular text in that it may be stored, read, searched, and edited, with one important exception: a hypertext document contains links to other documents. For instance, selecting the word "hypertext" in a first document could call up secondary

BNSDOCID: <WO_ 9907007A2_I_>

documents to the user's computer screen, including, for example, a dictionary definition of "hypertext" or a history of hypertext. These secondary documents would in turn include connections to other documents so that continually selecting terms in one documents after another would lead the user in a free-associative tour of information. In this way, hypertext links or "hyperlinks" can create a complex virtual web of connections.

Hypermedia include hypertext documents which contain links not only to other pieces of text, but also to other media such as sounds, images and video. Images themselves can be selected to link to sound or documents.

The standard language the Web uses to create and recognize hypermedia documents is the HyperText Markup ("HTML") Language. This language is loosely related to, but technically not a subset of, the Standard Generalized Markup Language ("SGML"), a document formatting language used widely in some computing circles. Web documents are typically written in HTML and are nothing more than standard text files with formatting codes that contain information about layout (text styles, document titles, paragraphs, lists) and hyperlinks.

HTML is limited to display functions (text, animation, graphics, and audio) and form submission. Inherent in the basic HTML protocol (set of software-encoded rules governing the format of messages that are exchanged between computers) are design limitations that prevent the implementation of the type of program functionality that is commonly used in today's desktop computers. Because of this limitation, it is impossible to enhance HTML documents with the features necessary to support the Internet applications currently being sought by industry and consumers.

Sun Microsystems and Netscape Communications have attempted to introduce additional program functionality through the implementation of Sun's Java programming

BNSDOCID: <WO__ 9907007A2 I >

language as a browser plug-in. Microsoft, in addition to supporting Java, has introduced Active-X as a browser plug-in. Currently, after two years of development, both Java and Active-X are still failing to deliver working software. These languages are plagued by software errors that crash the newer operating systems. Even the developers of these languages, Sun and Microsoft, are having major problems with Java and Active-X applications that they have written to demonstrate the capabilities of the languages.

Java, Active-X and almost all other programming languages use a programming paradigm based on the "C" programming language introduced by AT&T in the 1970's to develop the UNIX operating system. Although well suited for operating system development, "C" has two major drawbacks for Internet content delivery. The first drawback is that an entire program must be loaded before anything can be executed using the program. Because Internet content is open-ended, most applications can become very large, making downloading impractical due to the limited bandwidth of most Internet connections. Typical Java and Active-X applications take several minutes to download before they start running. The second drawback using programs based in "C" is that content development is very inefficient. "C" and programming languages based on the "C" paradigm are currently used to produce software tools and utilities such as word processors, spreadsheets, and operating systems. Software developers can assign a large number of programmers to a long-term project, with the knowledge that repeated sales of the product to existing users will recover the large investment expenditure. However, developers of Internet content cannot afford this type of approach and have fallen back upon the very simple HTML protocol to economize development.

However, even HTML is inadequate to enable or facilitate the conducting of interactive programming on the Internet. Although well adapted to passively providing multimedia information, HTML is extremely limited in active and interactive software use and

development. Thus, companies seeking to conduct commercial activities on the Internet are seeking a programming tool or information-exchange methodology to replace HTML or to provide major enhancement to that language. In contrast to applications programs based on the "C" paradigm, Internet applications programming is generally not subject to multiple uses. Rather an Internet program is consumed: the program is used only once by the typical user. Utilities-type software developed for individual computer use, such as word processors, spread sheets, E-mail, etc., can be sold at higher prices because the software is used again and again by any individual.

Programming languages based on the "C" programming paradigm are awkward for programming interactions involving the computer's display screen. For example, a Microsoft program for the manipulation of sprites (graphic images that can move over a background and other graphic objects in a nondestructive manner) requires over 800 lines of "C" code and over 290 lines of assembler code. The same program written in the TenCORE language requires only six lines.

TenCORE is a modular programming language designed in the early 1980's to facilitate the transfer of information between disk drives and RAM, particularly for the delivery of interactive instruction ("Computer-Based Training"). At that time, disk drives were all slow and RAM was inevitably small. In adapting to these hardware limitations, TenCORE introduced the use of small individual code modules for performing respective functions, thereby enabling efficient performance as each code module loaded a single interaction from the slow disk drives. Programming in the form of substantially independent code modules is open-ended, a necessary characteristic for educational software requiring the delivery of whatever remedial instruction is required by the individual student. TenCORE utilizes a program called an "interpreter" that implements all basic required functions efficiently. The

interpreter has no content itself. The content comes from an unlimited number of small pseudocode modules which are loaded into the system memory and issue commands to the interpreter. In writing a program, the programmer's involvement is reduced to simply issuing commands, leaving all the difficulties of implementation to the interpreter.

In contrast to TenCORE, "C" type programming includes a series of pre-written code libraries which are linked together as they are compiled. In theory, when a new microprocessor is designed, only the code libraries need to be rewritten and then all programs can be re-compiled using the new libraries. However, over the past twenty years the number of these code libraries has grown to a point where the original concept of rewriting for different microprocessors is no longer practical because of the difficulty in compensating for subtle differences between microprocessors. What remains is a complex and difficult programming syntax that takes years to master and is very difficult to debug.

Objects of the Invention

A general object of the present invention is to provide a method for communicating over a computer network.

A more particular object of the present invention is provide a method for enabling or facilitating the conducting of interactive programming over a computer network such as the Internet.

A related object of the present invention is to provide a method for communicating software over a computer network, to enable an increased degree of interactive computer use via the network.

Another object of the present invention is to provide a computing system for enabling or facilitating the conducting of interactive programming over a computer network such as the Internet.

It is a further object of the present invention to provide a computing system for communicating software over a computer network, to enable an increased degree of interactive computer use via the network.

These and other objects of the present invention will be apparent from the drawings and descriptions herein.

Summary of the Invention

The present invention is directed to a computerized system and to an associated method for optimally controlling storage and transfer of computer programs between computers on a network to facilitate interactive program usage. In accordance with the method, an applications program is stored in a nonvolatile memory of a first computer as a plurality of individual and independent machine-executable code modules. In response to a request from a second computer transmitted over a network link, the first computer retrieves a selected one of the machine-executable code modules and only that selected code module from the memory and transmits the selected code module over the network link to the second computer.

In one important field of use of the invention, the first computer is a server computer on a network. The second computer may be a user computer or a secondary server. Alternatively, the two computers may be two individual user computers on the network.

Where the second computer is a user computer, the request for the executable code module arises because the user needs the code module to perform a desired function in the applications program. The user's computer does not have all of the code modules of the applications program and obtains new code modules only as those modules are needed. Because executable code of the applications program is broken down into individually executable modules, the program can be downloaded piecemeal, module by module, as the

individual modules are needed by the user's execution of the program. Accordingly, the user need not wait for the entire program to be downloaded before the user can start using the program.

Where the first computer and the second computer are a primary server and a secondary server, respectively, the present invention allows the primary server, when is too busy to handle a user request or task, to hand that request or task off to the secondary server. When a primary server receives a user request or task that the server cannot handle due to load, that request or task is forwarded preferably to the least busy secondary server available. The secondary server then processes the user request and responds to the client/user directly. If the secondary server does not have a code module, data file, or other resource required to handle the user request, the required code module, data file, or other resource can be transferred to the secondary server from the first server. This shunting of user requests contemplates the utilization of multiple secondary servers located on different Internet connections. The present invention thus eliminates most bandwidth and server load issues on the Internet and other computer networks.

Of course, a server computer must shed its load before the server reaches its full capacity, thereby leaving enough system resources (processor, memory, etc.) free to fulfill any requests from a secondary server for resources necessary for the secondary server to fulfill a shunted user request or task.

If a secondary server is unable to contact the client machine, the secondary server can forward the user request to another secondary server for processing, since the other secondary server may be able to reach the client machine directly.

A secondary server may also serve as a backup server. In that case, the secondary server immediately requests all code modules, data files, and other resources from the primary

server instead of waiting for a client request that requires one of those resources. When a client or secondary server makes a request of the primary server and discovers it to be inaccessible, the request can then be made of a backup server. In this way, if a primary server were to fail, the information that was on the primary server would still be accessible through the backup server. To facilitate the maintenance of up-to-date resources on backup servers, the primary server sends notification packets to each of the backup servers whenever a resource is created or updated. The backup servers can then request the new, updated copy of the resource from the primary server, thereby remaining up to date with the primary server at all times.

In order to optimize processing of user requests, a primary server stores in its memory a list of secondary servers on the network, the list including response times and load statistics (processor usage, etc.) for the respective secondary servers. The primary server periodically updates these response times, by sending echo packets to the secondary servers and measuring delays between the sending of the echo packets and a receiving of responses to the echo packets from the respective secondary servers. The server also periodically updates the load statistics by requesting the current load statistics from the secondary servers. For processing a recently received user request, the primary server scans the list in memory and selects the secondary saver having the lightest load and shortest response time.

For security purposes, code modules, as well as other resources such as data files, are transmitted in encrypted form. Encryption is accomplished by placing the code modules, files, documents or other transmittable resources in the data area of an encryption packet. When an encrypted packet is received, it is decrypted and the code modules, files, documents or other transmittable resources subsequently handled pursuant to normal procedures. The encryption/decryption process can be handled by a plug-in code module. Any number of plug-

in encryption code modules may exist in a data and code transfer system in accordance with the invention at any one time. The header of an encryption packet contains an indication of which plug-in encryption code module must be used for decryption purposes. Encryption/decryption code modules can be delivered real time by a code module exchange protocol.

To further enhance the security of the system, the primary server stores a list of user authentification codes in its memory. Upon receiving a request, e.g., for a machine executable code module, from another computer, the primary server compares a user authentification code in the request with the stored list of user authentification codes. The primary server proceeds with retrieving and transmitting a selected machine-executable code module only if the user authentification code in the request matches a user authentification code in the stored list. Where an incoming request for a machine-executable code module is contained in an encryption packet, the server decrypts the encryption packet prior to the comparing of the user authentification code in the request with the list of user authentification codes in the memory. Encrypting and decrypting of encryption packets, as well as the checking of user authentification codes, may be performed by the secondary server(s). Whatever programming or data required for carrying out these security measures can be transmitted from the primary server in accordance with the present invention.

The present invention provides for the updating of an applications program in users' machines. When a user sends a request for a code module to a server, the request includes a specification of the version of the program code sought. The server processing the request checks whether the requested version is the latest version available. When a newer version of a requested code module is available, the server informs the user and inquires whether the user could use the newer version of the requested module. The user could then send a request for

the updated version of the desired code module.

If the updated version of the desired code module is incompatible with older versions of other code modules, the older version of the desired code module will be transmitted from the server to the user. If the older version is not available, the user may have to request transmission of the newer versions of several additional modules.

Generally, it is contemplated that machine-executable code modules are written in a user-friendly programming code such as TenCORE. In that case, a software-implemented interpreter unit of the user computer translates the code modules from the programming code into machine code directly utilizable by the user computer. When such an interpreter is used, the interpreter itself can be updated in the same fashion as an application program or code module. For purposes of updating the interpreter, the interpreter is treated as a single large code module.

In a related method in accordance with the present invention for optimally controlling storage and transfer of computer programs between computers on a network to facilitate interactive program usage, a portion of an applications program is stored in a first computer. The applications program comprises a plurality of individual and independent machine-executable code modules. Only some of the machine-executable code modules are stored in the first computer. This method includes executing at least one of the machine-executable code modules on the first computer, transmitting, to a second computer via a network link, a request for a further machine-executable code module of the applications program, receiving the further machine-executable code module at the first computer from the second computer over the network link, and executing the further machine-executable code module on the first computer.

To further facilitate program transfer on the network, the first computer may conduct

an investigation into server availability. Pursuant to this feature of the invention, a request is sent from the first computer (e.g., a user) to a further computer (e.g., a primary server) for a list of servers (e.g., secondaries) on said network. After transmission of the list of server to the first computer, response times for the server are determined by (a) sending echo packets from the first computer to the servas, (b) measuring, at the first computer, delays between the sending and receiving of the echopackets, (c) querying the servers as to current server load (processor usage, etc.) The request for a further machine-executable code module is sent to the server having the lightest load and shortest measured response time. The list of servers can be cached in memory by the first computer to facilitate further access to the information in the event that the server from which the list was requested becomes unavailable or is too busy to handle the request.

In the event that there has been an update in a requested code module, a request from the first computer for a particular version of the code module may trigger an inquiry as to whether the first computer could use the updated version of the code module. If so, the first computer transmits a second requests, this time for the updated version of the desired code module.

Where the two computers are both user machines, the method of the present invention facilitates interactive processing. Each computer executes one or more selected code modules in response to the execution of code modules by the other computer. Thus, both computers store at least some of the machine-executable code modules of the applications program. Occasionally one computer will require a code module which it does not have in present memory. Then the one computer can obtain the required code module from the other computer. If the other computer does not have the required module, a request may be made to a server on which the applications program exists in its entirety.

BNSDOCID: <WO 9907007A2 I >

Where the machine-executable code modules are written in a user-friendly programming code, each user computer includes an interpreter module implemented as software-modified generic digital processing circuits which translates the code modules from the programming code into machine code utilizable by the respective computer.

In accordance with a feature of the present invention, the machine-executable code modules each incorporate an author identification. The method then further comprises determining, in response to an instruction received by a user computer over the network and prior to executing the one of the machine-executable code modules on the user computer, whether the particular author identification incorporated in the one of the machine-executable code modules is an allowed identification. The user computer proceeds with code module execution only if the particular author identification is an allowable identification. Generally, the instruction determining whether a code module is written by an allowed author is a list of blacklisted authors.

The author identification feature of the invention severs to prevent an author from creating a virus or other malicious program and distributing it using the code module exchange protocols of the present invention. All compilers supporting the coding of individual machine-executable code modules in accordance with the present invention will incorporate a respective author identification code into each machine-executable code module. The author identification is unique to each author. When a malicious program is discovered, the identification of the author may be distributed throughout the network to blacklist that author and prevent the execution of code modules containing the blacklisted author's identification. As an additional advantage, this feature will discourage users from distributing illegal copies of the authoring package, since the respective users will be held responsible for any malicious programs written under their author identification.

The storing of applications program modules in a user computer may include caching the code modules in a nonvolatile memory of the user computer.

It is to be noted that the present invention permits the transmission during user computer idle time of code modules whose use is anticipated. A request from the user computer is transmitted to a server or other computer for a machine-executable code module during an idle time on the user computer.

A computing system comprises, in accordance with the present invention, digital processing circuitry and a nonvolatile memory storing general operations programming and an applications program. The applications program includes a plurality of individual and independent machine-executable code modules. The memory is connected to the processing circuitry to enable access to the memory by the processing circuitry. A communications link is provided for communicating data and programs over a network to a remote computer. A code module exchange means is operatively connected to the memory and to the communications link for retrieving a single code module from among the machine-executable code modules and transferring the single code module to the remote computer in response to a request for the single code module from the remote computer.

As discussed above, the computing system of the present invention may be a server computer on the network. The server's memory may contain a list of secondary servers on the network, the list including response times for the respective secondary servers, The computing system then further comprises a detection circuit, generally implemented as software-modified generic digital processing circuitry, for detecting an overload condition of the computing system. A server selector, also generally implemented as software-modified generic digital processing circuitry, is operatively connected to the detection circuit, the memory and the communications link for determining which of the secondary servers has a shortest response

time and for shunting an incoming user request to the secondary server with the shortest response time when the overload condition exists at a time of arrival of the user request.

Thus, the remote computer transmitting a code-module request to the computing system may be a secondary server to which a user request has been shunted. The requested single code module is required for enabling the remote computer to process the user request. On behalf of the computing system.

Pursuant to another feature of the present invention, the computing system further comprises an updating unit, preferably implemented as software-modified generic digital processing circuitry, operatively connected to the memory and the communications link for (1) periodically sending echo packets to the secondary servers, (2) measuring delays between the sending of the echo packets and a receiving of responses to the echo packets from the respective secondary servers, and (3) updating the response times in the list in accordance with the measured delays.

For security purposes, the memory of the computing system may contain a stored list of user authentification codes. The computing system then includes a comparison unit, preferably implemented as software-modified generic digital processing circuitry, for comparing a user authentification code in an incoming request with the list of user authentification codes in the memory and for preventing code-module retrieval and transmission in the event that the user authentification code in the request fails to correspond to any user authentification code in the list.

Where incoming requests for code modules are contained in encryption packets, the computing system further comprises a software-implemented decryption circuit connected to the communications link and the comparison unit for decrypting the encryption packet prior to the comparing of the user authentification code in the request with the list of user

authentification codes in the memory.

In accordance with another feature of the invention, the computing system includes means for determining whether a requested code module has an updated version and for responding to the request with an invitation to the remote computer to accept the updated version of the requested code module.

It is contemplated that the machine-executable code modules are written in a user-friendly programming code. Where the computing system uses the applications code itself, the computing system further comprises an interpreter for translating the programming code into machine code directly utilizable by the processing circuitry. The interpreter may take the form of generic digital processing circuit modified by programming to perform the translation function.

A computing system (e.g., a user computer on a network) also in accordance with the present invention comprises a memory storing a portion of an applications program having a plurality of individual and independent machine-executable code modules, only some of the machine-executable code modules being stored in the memory. A digital processing circuit is operatively connected to the memory for executing at least one of the machine-executable code modules. A communications link is provided for communicating data and programs over a network to a remote computer. A code module exchange unit is operatively connected to the memory and to the communications link for communicating with a remote computer (e.g., a server on the network) via a network link to obtain from the remote computer a further machine-executable code module of the applications program. The digital processing circuitry is operatively tied to the code module exchange unit for executing the further machine-executable code module upon reception thereof from the remote computer.

When a client or user machine is not running at full capacity (processor idle time is over

a given threshold and network traffic is under a given threshold), that machine can look for other client machines on the same virtual network that may be in the midst of a processor-intensive task and take on some of the load. If necessary, the code modules to handle that task can be transferred to the idle client machine. It is possible for clients working together on the same project to communicate with each other using a custom sub-protocol. This client-side distributed processing significantly improves the performance of processor-intensive tasks.

The present invention provides a programming paradigm which addresses the Internet content developer's specific needs. A software system and computer communications method in accordance with the present invention delivers rapidly over the Internet, provides a practical programming paradigm that supports rapid economical development of content, and facilitates new capabilities in Internet software and systems management.

TenCORE, a modular programming language designed to use small efficient code modules for facilitating program transfer between disk drives and central processing units of desktop computers, may be easily modified to carry out the present invention. Minimal modifications require the adapting of the transfer capabilities to work over network links.

The present invention arises in part from the realization by the inventors that the problems facing the developers of TenCORE in 1980 are the same problems that software designers face today when dealing with the Internet and its limited bandwidth and slow connections. It was perceived that Internet applications must be open-ended and programming must be delivered rapidly in spite of bandwidth limitations. Thus, the solution provided by TenCORE is useful in solving today's problems with interactive software on the Internet. The modular programming of TenCORE enables rapid Internet performance because a single TenCORE Net programming module can be quickly downloaded over the Internet. The TenCORE Net programming modules are TenCORE programming modules which are

modified to enable downloading from Internet servers instead of from a microcomputer's disk drive.

TenCORE and TenCORE Net are interpreted languages, i.e., they serve to translate into machine language pseudocode programs and to perform the indicated operations as they are translated. "Pseudocode" is unrelated to the hardware of a particular computer and requires conversion to the code used by the computer before the program can be used. Once the TenCORE Net interpreter is installed on a computer equipped with an Internet connection, clicking with a mouse on a conventional World-Wide Web hyperlink that points to a TenCORE Net application automatically bypasses the Internet browsing software and launches the application. Because only the required modules are sent over the Internet and because the TenCORE Net modules are very small, Internet performance is very fast.

Brief Description of the Drawing

Fig. 1 is a block diagram of a computer network with various servers and user computers, which transmit executable programs to one another pursuant to the present invention.

Fig. 2 is a block diagram of a primary server in accordance with the present invention.

Fig. 3 is a block diagram of a user computer in accordance with the present invention.

Fig. 4 is a flow chart diagram illustrating a maintenance loop in the operation of selected program-modified processing circuits in the primary server of Fig. 2 and the user computer of Fig. 3.

Fig. 5 is a flow chart diagram illustrating a loop for processing an incoming data packet in the operation of selected program-modified processing circuits in the primary server of Fig. 2 and the user computer of Fig. 3.

Figs. 6A and 6B are a flow chart diagram illustrating further operations relating to the

processing of an incoming request for a code module in accordance with the present invention.

Fig. 7 is a flow chart diagram illustrating a subroutine for processing an incoming resource request packet in the operation of selected program-modified processing circuits in the primary server of Fig. 2 .

Fig. 8 is a flow chart diagram illustrating a maintenance loop in the operation of selected program-modified processing circuits in the primary server or a secondary server of Fig. 2.

Fig. 9 is a flow chart diagram illustrating a subroutine for processing an incoming resource request packet in the operation of selected program-modified processing circuits in the primary server of Fig. 2 .

Fig. 10 is a flow chart diagram illustrating a maintenance loop in the operation of selected program-modified processing circuits in the primary server or a secondary server of Fig. 2

Description of the Preferred Embodiments

As illustrated in Fig. 1, a computer network comprises a plurality of user computers 12 operatively connected to a primary server computer 14 via a complex of network links 16. The primary server 14 stores, in an area 18 of a nonvolatile memory 20 (Fig. 2), an applications program which may be desired for use on one or more user computers 12. The term "applications program" as used herein refers to any executable collection of computer code other than operating systems and other underlying programming for controlling basic machine functions.

As further illustrated in Fig. 1, the network includes a plurality of secondary servers 22 which are available for assisting primary server 14 in responding to requests from user computers 12. More particularly, as shown in Fig. 2, primary computer 14 includes an

overload detector 24 which continually monitors a queue of jobs or tasks which primary server 14 has to perform. Overload detector 24 determines whether the number and size of tasks in the queue has exceeded a predefined threshold. Once that threshold is reached, overload detector 24 signals a secondary server selector 26 which accesses an area 28 of memory 20 to identify a secondary server 22 which is least busy. To that end, memory area 28 contains a list of secondary servers 22 as well as measured response times for the respective servers.

The response times in the secondary-server list in memory area 28 are periodically measured by dispatching echo packets to each secondary server 22 and measuring the delays between the transmission of the respective echo packets from primary server 14 and the receipt of responses from the respective secondary servers 22. To accomplish the measurement, primary server 14 and, more particularly, a processing unit 30 thereof contain an echo packet dispatcher 32 and a response-time monitor 34. Upon transmitting an echo packet to a secondary server 22 via a network communications interface 36 at primary server 14, dispatcher 32 notifies response-time monitor 34 as to the transmission of a packet and as to the secondary server 22 to which the transmission was made. Monitor 34 counts out the time between the transmission of the echo packet and the arrival of a response from the respective secondary server 22 via network links 16 and communications interface 36. A response-time update unit 38 is connected to response-time monitor 34 and to memory area 28 for writing updated response times in the secondary server list stored in memory area 28.

The applications programs store 18 in memory 20 contains a multiplicity of individual and independent machine-executable code modules. Accordingly, a user computer 12 working with the applications program need not be loaded with the entire program in order to begin processing operations. In the event that the user's execution of the applications program requires a code module not contained in the user's computer memory, a request is transmitted

from the user computer 23 over network links 16 to primary server 14. If primary server 14 is not overloaded, it retrieves the requested code module from program store 18 and transmits it over communications interface 36 and network links 16 to the user computer which made the request.

Processing unit 30 of primary server 14 includes a code-module exchanger 40 which processes incoming requests for code modules of the applications program or programs stored in memory area 18. Exchanger 40 cooperates with a version detector 42 which consults the applications program store 18 to determine whether a requested version of a particular code module is the latest version in store 18. If not, version detector 42 and code module exchanger 40 transmit an inquiry to the resting computer as to whether the latest version of the desired code module is utilizable by the requester. If the newer version of the desired code module can be used in place of the older version, the newer version is transmitted over communications interface 36 and network links 16.

Code modules, as well as other resources such as data files, may be transmitted in encrypted form for security purposes. Encryption is accomplished by placing the code modules, files, documents or other transmittable resources in the data area of an encryption packet. When an encrypted packet is received by processing unit 30 via communications interface 36, the packet is forwarded via generic processing circuits 50 to an encryption/decryption unit 44 for deciphering. Encryption/decryption unit 44 consults a memory area 46 containing a plurality of possible encryption keys and selects an encryption key identified by header information in the encryption packet containing the user request. Upon decryption of the incoming encryption packet, the user request or code module exchange packet contained therein is forwarded to code-module exchanger 40 for processing. The encryption/decryption process can be handled by a plug-in code module performing the

functions of encryption/decryption unit 44 and memory area 46. Any number of plug-in encryption code modules may exist in a data and code transfer system at any one time. The header of an encryption packet contains an indication of which plug-in encryption code module must be used for decryption purposes.

To further enhance the security of a computer network which has protocols for the exchange of executable code modules, primary server 14 stores a list of user authentification codes in a memory area 48. In response to a request, e.g., for a machine-executable code module, from another computer, comparator 52 in processing unit 30 compares a user authentification code in the request with the list of user authentification codes stored in memory area 48. Code module exchanger 40 proceeds with retrieving and transmitting a selected machine-executable code module only if the user authentification code in the request matches a user authentification code in the stored list. Where an incoming request for a machine-executable code module is contained in an encryption packet, encryption/decryption unit 44 deciphers the encryption packet prior to the comparing by comparator 52 of the user authentification code in the request with the list of user authentification codes in memory area 48. Where a user request is relayed to a secondary server 22 chosen by selector 26, the secondary server incorporates an encryption/decryption unit and an authentification-code comparator for encrypting and decrypting of encryption packets and the checking of user authentification codes, may be performed by the secondary server(s). Whatever programming or data required for carrying out these security measures can be transmitted from primary server 14 to the selected secondary server 22.

As illustrated in Fig. 3, a processing unit 54 of a user computer 12 includes a code-module exchanger 56 which generates requests for desired code modules of an applications program as those code modules are required during execution of the applications program by

the user. The code-module requests are transmitted to code-module exchanger 40 of primary server 14 via a network communications interface 58 at user computer 12, network links 16 (Fig. 1) and communications interface 36 at primary server 14.

In a security enhanced system, processing unit 54 of user computer 12 includes a encryption/decryption unit 60 which inserts code-module request packets, as well as other information transfer packets, into the data areas of respective encryption packets whose encryption keys are selected from a plurality of keys stored in an area 62 of a nonvolatile memory 64 of user computer 12. Again, the encryption/decryption process at user computer 12 can be handled by a plug-in code module performing the functions of encryption/decryption unit 60 and memory area 62. The header of an encryption packet generated by encryption-decryption unit 60 contains an indication of which plug-in encryption code module at primary server 14 must be used for decryption purposes.

In a further security enhancement used to protect the computing system of Fig. 1 in general and user computer 12 in particular from computer viruses and other malicious programming, processing unit 54 is provided with an author identification comparator 66 which accesses an author identification list 68 in memory 64. Author identification list 68 is periodically updated by author identification comparator 66 and other function-converted blocks in generic processing circuits 70 in response to incoming instructions from primary server 14. Author identification list 68 contains a list of allowed authors or, perhaps more efficiently, a list of authors who have been blacklisted owing to their passing of viruses or other malicious programming onto the network. Prior to the use of an incoming machine-executable code module. The author identification in a packet header is checked by comparator 66 to determine that the author of the particular code module has not been blacklisted. If the author is allowed to produce executable code modules for user computers 12, processing of the

incoming code module proceeds normally. If, on the other hand, the author of the incoming code module has been blacklisted, then the code module is never executed and may be discarded prior to storage in an applications program store 72 in memory 64.

As discussed above with reference to Figs. 1 and 2, primary server 14 may occasionally hand off an incoming user request to a secondary server 22, depending on the load condition of the primary server and the secondary servers. Generally, this handing off of responsibility for responding to users' requests is transparent to the users, i.e., the processing of users' requests proceeds without knowledge or intervention by the users. It is also possible for user computers 12 to take over selection of a secondary computer 22 from primary computer 14. To that end, processing unit 54 of user computer 12 is provided with a secondary server selector 74 which accesses a list 76 of secondary-servers in memory 64. Generally, selector 74 selects a secondary server 22 with a smallest response time relative to the respective user computer 12. To that end, processing unit 54 further includes an echo packet dispatcher 78, a response-time monitor 80 and a response-time update unit 82. Dispatcher 78 sends echo packets to secondary servers 22 via communications interface 58 and network links 16 (Fig. 1), while monitor 80 determines the delays of responses from the various secondary servers. Update unit 82 corrects response time data in list 76 in accordance with measurements carried out by dispatcher 78 and monitor 80. It is contemplated that the updating of secondary-server response times in list 76 is implemented only when user computer 12 requires a user module or other resource from primary server 14 and that server is too busy to handle the user request.

Processing unit 54 of a user computer 12 has a distributed processing circuit 84 which enables processing unit 54 to share the processing of large tasks with other user computers in the network. Thus, when a user computer is not running at full capacity (processing unit 54 is more than 50% idle and there is no network traffic), that distributed processing circuit 84

looks for other user computers 12 that may be in the midst of a processor-intensive task and enable transfer of some of the load to the processing unit 54 of the respective user computer 12. If necessary, the code modules to handle that task can be transferred to the idle user computer 12. This client-side distributed processing significantly improves the performance of processor-intensive tasks.

Processing unit 54 of a user computer 12 also contains a metered delivery and billing circuit 86 which controls access to content which must be paid for. Credit registers 88 in memory 64, accessible by circuit 86, store credits which the particular user has against respective accounts. When credit in an account maintained or monitored by primary server 14 is low, circuit 86 may arrange for the transfer of more credit from primary server 14 to the particular user. Metered delivery and billing circuit 86 includes a billing method to pay for the credit. Generally, credit requests and responses thereto should be encrypted.

Processing unit 54 of a user computer 12 additionally includes a unidirectional data submission and collection circuit 90 which accesses data files 92 in memory 64 for purposes of uploading those data files to primary server 14 or to a selected secondary server 22. Data submission and collection circuit 90 is operative when the user computer does not need to read the data back from the server. This circuit is particularly useful for on-line orders, form submission, and data collection (including statistical data) among other things.

Generally, packets that contain data to be written must be sent directly to the primary server 14 and cannot be shunted. This prevents conflicts between different versions of the resource that was written to. Data written back to the server should require user authentification. User authentification should be used even if the write-back will be done only by a program under author control. In that case, user identification codes and passwords can be built into the program. The reason for this is to prevent another author from writing a

program that would write back to the same data and possibly corrupt it.

Applications program code modules stored in memory area 18 (Fig. 2) and module store 72 (Fig. 3) are written in TenCORE, a modular programming language originally designed to use small efficient code modules for facilitating program transfer between disk drives and central processing units of desktop computers. TenCORE is easily modified, for example, to adapt the code transfer capabilities to operate over network links. The program so modified for use on user computers 12, primary server 14 and secondary servers 22 may be called "TenCORE Net."

TenCORE Net uses small individual code modules for performing respective functions, thereby enabling efficient performance as each code module is downloaded a single interaction from primary server 14 or a selected secondary server 22. Programming in TenCORE Net is open-ended, so that a user computer 12 executes instructions of an applications program when that applications program is only partially stored in program store 72 of memory 64.

The code modules held in memory store 72 are in a user-friendly pseudocode language which must be translated or interpreted for direct machine use. To that end, processing unit 54 includes a program or programmed interpreter circuit 94 that implements all basic required functions efficiently. The interpreter has no content itself. Content is derived from a potentially unlimited number of small pseudocode modules which are loaded into memory store 72 and which effectively issue commands to interpreter 94. In writing a program, the programmer's or author's involvement is reduced to simply issuing commands, leaving all the difficulties of implementation to the interpreter.

TenCORE may be modified in two ways to enable network use. First, a subroutine or set of instructions may be inserted before each call line of computer code for checking whether the code intended for execution exists in applications program memory area 72. If not, code

module exchanger 56 is instructed to obtain the required code module from primary server 14. Once the required code module has been downloaded into memory area 72, it is called and translated by interpreter circuit 94 prior to execution by processing circuits 70.

Through the use of modular applications programs and code-module exchangers 40 and 56, the control of computer program storage and transfer between computers 12, 14, 22 on a network is optimized, thereby facilitating interactive program usage. An applications program stored in nonvolatile memory area 18 of primary server 14 may be transferred to various user computers 12 in modular fashion. In response to a request transmitted over network links 16 from a user computer 12 or a selected secondary server 22, primary server 14 retrieves a selected machine-executable code module and only that selected code module from memory area 18 and transmits the selected code module over network links 16 to the user computer or secondary server.

Where the applications program is, for instance, a drawing or painting program, a user computer 12 may be instructed to draw a three-dimensional geometric figure such as a truncated pyramid. If processing circuits 70 discover that the applications program in program store 72 is missing a code module required for performing that task, code module exchanger 56 requests that the requisite module be transferred from primary server 14. In response to that request, version detector 42 may find that a later version of the desired module exists and inquire of code-module exchanger 56 whether the later version would be acceptable for use by the respective user computer.

Thus, user computers 12 do not have all of the code modules of the applications program and obtain new code modules only as those modules are needed. Because executable code of the applications program is broken down into individually executable modules, the program can be downloaded piecemeal, module by module, as the individual modules are

needed by the user's execution of the program. Accordingly, the user need not wait for the entire program to be downloaded before the user can start using the program.

Similarly, a selected secondary server 22 can begin to process a transferred or shunted user request and respond to the client/user directly, without having the entire applications program and other resources relating to the handling of the program's distribution by the primary server 14. If the selected secondary server does not have a code module, data file, or other resource required to handle the user request, the required code module, data file, or other resource can be transferred to the secondary server from the primary server. Secondary servers 22 are thus provided with code module exchangers similar to exchangers 40 and 56.

Of course, primary server 14 must shed its load before that server reaches its full capacity, thereby leaving enough system resources (processor, memory, etc.) free to fulfill any requests from a secondary server for resources necessary for the secondary server to fulfill a shunted user request or task.

In security sensitive networks, secondary servers 22 are equipped with encryption/decryption units like unit 44, as well as authentification comparators 52. Where an incoming request for a machine-executable code module is contained in an encryption packet, the secondary server decrypts the encryption packet prior to the comparing of the user authentification code in the request with a list of user authentification codes in memory. Whatever programming or data required for carrying out these security measures can be transmitted from primary server 14.

If a secondary server is unable to contact the client machine, the secondary server can forward the user request to another secondary server for processing, since the other secondary server may be able to reach the client machine directly.

Code module exchanger 56 of processing unit 54 facilitates interactive processing on a

network. Each user computer 12 executes one or more selected code modules in response to the execution of code modules by the other computer. Thus, both computers store at least some of the machine-executable code modules of the applications program. Occasionally, one computer will require a code module which it does not have in present memory. Then the one computer can obtain the required code module from the other computer. If the other computer does not have the required module, a request may be made to a server on which the applications program exists in its entirety.

Thus, clients or users on a network are able to exchange code modules with one another. If a first user and a second user are engaged in an interactive whiteboarding session and the first user starts drawing with a tool that the second user does not have in her version of the whiteboarding program, the code module or modules for that tool can be transferred automatically from the first user's computer to the second user's computer.

It is to be noted that code module exchanger 56 may be instructed to initiate the transmission during user computer idle time of code modules whose use is anticipated. A request from the user computer 12 is transmitted to primary server 14 or other computer for a machine-executable code module during an idle time on the user computer. Resource requests that are generated for idle-time downloading should be flagged as such, so that code module exchanger 56 can assign different priorities from standard requests for load balancing and distribution purposes. The status of a resource request can be upgraded to real time from idle time in the event that the user attempts to access the associated section of the application.

It is to be noted that the various dedicated function blocks of processing units 30 and 54 are generally and preferably implemented as software-modified generic digital processing circuits. Accordingly, code-module exchanger 40 and 56 are characterizable as protocols for the exchange of code modules between a server 14 or 22 and a user computer 12. This code

module exchange protocol is considered a subprotocol of a Modularized Code Master Protocol ("MCMP") which handles load distribution, user authentification and encryption. Load distribution is more particularly handled in primary server 14 by processor overload detector 24 and secondary-server selector 26, while user authentification and encryption are handled in primary server and secondary servers 22 by comparator 52 and encryption/decryption unit 44.

The MCMP has four subprotocols, namely, the Code Module Exchange Protocol ("CMXP"), the Uni-Directional Data Submission and Collection Protocol ("UDSCP"), the Metered Delivery and Billing Protocol ("MDBP"), and the Distributed Processing Protocol (DPP"). The Code Module Exchange Protocol is realized by code-module exchanger 40 in processing unit 30 of primary server 14 and secondary servers 22 and by code-module exchanger 56 in processing unit 54 of user computers 12. The Uni-Directional Data Submission and Collection Protocol is implemented by circuit 90 in user computers 12 and by corresponding non-illustrated program-modified processing circuitry in primary server 14. The Metered Delivery and Billing Protocol finds realization by circuit 86 in user computers 12 and by corresponding non-illustrated program-modified processing circuitry in primary server 14. The Distributed Processing Protocol takes the form of circuit 84 in processing unit 54 of user computers 12.

Fig. 4 illustrates operations undertaken by echo dispatcher 32, response-time monitor 34, and update unit 38 as well as other processing circuits of processing unit 30 of primary server 14 to maintain an updated list 28 of the availability of secondary servers 22 . The same steps may be performed by echo dispatcher 78, response-time monitor 80, and update unit 82 as well as other processing circuits of processing unit 54 of user computer 12 to obtain an updated list of secondary server response times. In an inquiry 100, echo dispatcher 32 or 78 or

generic processing circuits 50 or 70 query whether the time since the last echo testing is greater than a predetermined maximum period TMR. If the maximum period has been exceeded, echo packet dispatcher 32 or 78 transmits, in a step 102, an echo packet to the secondary server 22 which was tested the least recently. Response-time monitor 34 or 80 determines in an inquiry 104 whether a response has been received from the targeted secondary server 22. If a response has not been received and if a prespecified number of measurement attempts has not been exceeded, as determined at a decision junction 106, another echo packet is dispatched in step 102. If a response is received from the targeted secondary server 22, update unit 38 or 82 then records, in list 28 or 76, the time between the initial packet transmission by dispatcher 32 or 78 and the receipt of the echo packet by monitor 34 or 80. This recordation is effected in a step 108. If the number of attempts at secondary-server response-time measurement has exceeded the pre-specified number, as determined at decision junction 106, the server is marked in list 28 or 76 as being unavailable (step 110). Also a message or alert signal may be generated to inform a server overseer. If at query 100, it is determined that the time since the last echo testing is less than predetermined maximum period TMR, processing unit 30 or 54 investigates at 112 whether there is a packet in a MCMP incoming packet queue. If not, the maintenance loop of Fig. 4 is re-entered. If so, a packet processing operation 114 is executed. It should be noted that the MCMP incoming packet queue contains both packets received from the network, and packets placed into the queue by the MCMP protocol.

Figs. 6A and 6B illustrate operation 114 carried out under the MCMP protocol by processing unit 30 of primary server 14 or of a secondary server 22 selected for overflow handing. In an initial inquiry 124, overload detector 24 decides whether processing unit 30 is too busy to handle the incoming packet (which may be, for example, a user request for a code

module). If so, processing circuits 50 investigate the MCMP packet header at a junction 126 to determine whether the packet can be shunted to a secondary server 22. If so, a further investigation 128 is conducted to determine whether the incoming packet has already been shunted. If the packet was sent to the server directly by the originating computer (i.e., not shunted), secondary-server selector 26 accesses list 28 in a step 130 to find the secondary server 22 with the lightest load. At a subsequent inquiry 132, processing unit 30, and more particularly, server selector 26, determines whether the selected secondary server is suitable for transfer of responsibility for the incoming packet. If the selected secondary server is suitable, the packet is flagged as the result of a service hand-off and forwarded to the selected secondary server (step 134).

If the secondary server selected in step 130 is not suitable for a hand-off, for example, if the response time of the secondary server is greater than a predetermined maximum, as determined at inquiry 132, a query 136 is made as to whether an incoming packet is the result of a service hand-off. This inquiry 136 is also made if the packet cannot be shunted, as determined at decision junction 126.

If an incoming MCMP packet is the result of a service hand-off, as determined at query 136, processing circuits 50 undertake an investigation 138 as to whether the requested resource is available. If the resource is available, processing circuits 50 ask at 140 whether the resource has passed a pre-assigned expiration date. If so, a signal is transmitted in a step 142 to the source of resource to determine if a newer version of the resource is available. If a new version is available, as ascertained at a decision junction 144. a request for the newer version of the resource is transmitted to the source in a step 146. This request is flagged as non-shuntable and should be additionally flagged as a priority request.

Prior to the processing of an incoming packet, e.g., a user request for a code module,

processing unit 30 examines the header information in the incoming packet at an inquiry 150 to determine whether the packet contains user authentification information. Where user authentification information is found, the former encryption status of the packet is determined at 152. If the packet was not encrypted, a message is generated in a step 154 to report that the user authentification failed. If the incoming packet was encrypted, the MCMP header information is checked at 156 to determine if the source server is specified. If there is no source server specification, user authentification failure is reported in step 154. If there is a source server specified in the MCMP header information and if that source server is not the host server, as determined at a decision junction 158, an investigation 160 is conducted (by authentification code comparator 52) as to whether memory area 48 has a non-expired, cached copy of the user authentification data. If there is no non-expired, cached copy of the user authentification data in memory area 48, comparator 52 induces processing circuits 50 to obtain the user's authentification data from the source server and to store that data in memory area 48 (step 162). If a user's password contained in the MCMP packet header information does not match the cached password, as determined by comparator 52 in an evaluation 164 or if the user is not listed with the source server, as discovered by processing circuits 50 at a check point 166, the user authentification is reported as failed in a step 168. A report as to user authentification failure (step 154) is also made if the source server is the host server (decision junction 158) and if comparator 52 finds in an investigation 170 that the user authentification data in the packet header does not correspond to any user authentification data in memory area 48.

Once comparator 52 finds a match between the authentification code in an incoming packet and the user's authentification code in memory area 48, as determined in investigation 170 or in evaluation 164, or if the incoming packet did not contain a user authentification code,

then evaluation 171 determines whether the packet should be handled directly by the MCMP protocol, or by another protocol. If evaluation 171 determines that the packet should be handled by the MCMP protocol directly, then the packet is processed accordingly at a step 173 as illustrated in Fig. 5. If evaluation 171 determines that the packet should be handled by a specific protocol, then processing circuits 50 determine in a step 172 which protocol (e.g., CMXD, UDSCP, MDBP, DPP) is appropriate for handling the content of the packet. If a response is produced by processing unit 30 under the selected protocol or by the main MCMP protocol, as determined in an inquiry 174, that response is tranenutted in a step 176 to the client that originally made the request. If there was a service handoff, that is, if the packet was shunted to the host server, then the response will be transmitted to a computer other than the computer from which the host received the packet. In a step 178, processing unit 30 begins processing the next packet in the queue or waits for a new packet to arrive.

As shown in Fig. 5, processing operation 173 includes an initial inquiry 116 into the type of packet. If the packet is an encryption packet, encryptionldecryption unit 38 or 60 is activated in a step 118 to decrypt the packet using the appropriate decryption module or key. In a subsequent step 120, the packet encased in the data area of the encrypted packet is flagged as non-shuntable and placed back into the MCMP incoming packet queue. If it is determined at inquiry 116 that the packet is not an encryption packet, an MCMP status report indicated an unknown packet type is issue in a step 122 and the packet is discarded. The functionality of the MCMP protocol may be enhanced at a later time by enhancing the process illustrated in Fig. 5 to include conditions for additional types of packets.

The Code Module Exchange Protocol (CMXP) handles dynamic downloading of executable code, program version control, client-to-client module exchange, virus and malicious program protection, data uploading, idle-time downloading, and code module

caching. These functions are variously performed in servers 14 and 22 by code module exchanger 40 and version detector 42 and in user computer 12 by code module exchanger 56, author identification comparator 66, and unidirectional data submission and collection circuit 86, as well as by various nondesignated generic processing circuits 50 and 70. The server-side portion of the CMXP protocol, as implemented in part by code module exchanger 40, handles the delivery of code modules and supporting resources such as graphic images and fonts. Requests to the CMXP server and to code module exchanger 40 can reference a file, a portion of a file (such as a code module), or a portion of a code module or other supporting module. Because programs are broken down into separate code modules, these code modules can be delivered on an as-needed basis, eliminating the need to download an entire program before execution thereof can commence.

There are several ways of accommodating or incorporating an upgrade where programs are delivered piecemeal, module by module. If the old and new versions are completely compatible (for example, the new version was generated as the result a fix to a typographical error in a dialog box), the new modules can be merged with the old modules. Version information is stored on a per-file basis as well as a per-code-module basis. This means that code modules which were not changed in a version upgrade do not need to be downloaded again. If the old and the new versions are not compatible and cannot be merged, and the entire program has been cached locally or is still available from the server, the old version of the program can continue to execute until the next time the program is restarted. If the old and the new versions are not compatible and cannot be merged, and the old version of the program is no longer available in its entirety, the program should be immediately terminated and restarted using the new version code modules. The author of a program can override any of these update procedures by including a custom update module in the new version of the program,.

This code module is downloaded (if necessary) and executed whenever a version conflict arises. Then, an election can be made to perform any of the above procedures or a custom procedure such as remapping the contents of the program's memory area so that they can be used by the new version.

Code modules and associated resources are cached by both user computers 12 and secondary servers 22. The caching rules can be incorporated into the CMXP protocol and/or the applications program itself. This allows custom caching rules to be built into an application, thus providing for special caching schemes and hybrid applications. When a code module download is in progress, the number of bytes completed is stored in the cache along with the actual data downloaded. If a download is interrupted, it can resume where it left off at a later time.

Fig. 7 illustrates operations executed by digital processing circuits of processing unit 30 which are functionally modified in accordance with the CMXP protocol. The operations include a check on author identification. Generally, the author blacklist in memory area 68 of user computers 12 is transmitted to the user computers from a server which undertakes maintenance operations to keep the list updated (Fig. 8).

As illustrated in Fig. 7, processing circuits 50 inquire at 180 whether an incoming packet constitutes a request for a resource. An affirmative answer to this inquiry leads to a further inquiry 182, as to whether the requested resource is available for anonymous access. If the resource is restricted, a determination 184 is made as to whether the requesting user has rights to access the requested resource. If the user has no such rights, an "access denied" message is returned to the requester in a step 186. If the requested resource is available to the requesting party, processing circuits 50 determine at a decision junction 188 whether the requested resource contains executable code. An affirmative determination causes a query 190

as to the validity of the author's fingerprint. If the fingerprint or author identification is invalid, a message is generated for a responsible party in a step 192. The local copy of the resource is deleted in a subsequent step 194 and a message "Resource Distribution Prohibited" is transmitted to the requesting party in a step 196.

If in response to query 190 it is found that the fingerprint of the author of the requested resource is valid, then a check 198 is made as to whether the author is blacklisted. A blacklisting leads to deletion of the local copy of the resource in step 194 and the issuance of the message "Resource Distribution Prohibited" in step 196. If the author is not blacklisted, or if the requested resource does not contain executable code, as determined at decision junction 188, then the processing unit 30 queries at 200 whether the client already has an up-top-date copy of the resource. If the client or user already has the latest version of the resource, a message to that effect is transmitted in a step 202. If the client or user's copy of the resource is an older version, the requested resource is transmitted to the client in a step 204.

If an incoming packet does not constitute a request for a resource, as ascertained in response to at inquiry 180, an investigation 206 is made as to whether the packet constitutes a request to modify a resource. If so, and if the resource is not available for anonymous modification, as determined at a decision junction 208, then processing unit 30 queries at 210 whether the user has rights to modify the resource. If the user has no such rights, then a message "Access Denied" is returned to the requester in a step 212. If the resource is available for modification by anybody (decision junction 208) or by the particular user (query 210), the processing unit 30 makes the requested modification to the resource in a step 214 and notifies key secondary servers, in a step 216, of the change to the resource.

If an incoming packet does not constitute a request for a resource, as ascertained in response to inquiry 180, and is not a request to modify a resource, as determined in

investigation 206, then the processing unit 30 checks at 218 whether the packet is an update to a prohibition list, for example, a list of prohibited users or blacklisted authors. If the packet is such an update and is encrypted, as determined at decision junction 220, then the processing unit 30 determines at an inquiry 222 whether the packet was sent under the system user account. If so, the cached copy of the prohibition list is updated in a step 224 and all secondary servers are notified of the update in a step 226. If an incoming update request is not encrypted (decision junction 220) or is not sent under the system user account (inquiry 222), then an alert message is issued to an appropriate party in a step 228. In a step 230, a special status report is issued if an unknown packet type is received.

In a CMXP maintenance loop, shown in Fig. 8, for updating a list of blacklisted authors, processing unit 30 asks in an initial query 232 asks whether the time since the last list update is more than a predetermined number of hours. If that much time has passed, an attempt 234 is made to contact the next server upstream of the server conducting the inquiry. If that server cannot be contacted, as determined in a scan 236, a check 238 is made as to whether there are other servers available. If another server is available, an attempt 240 is made to contact that server. If no server can be contacted, the time is again checked, at 242. If a predetermined interval has lapsed since the last update, then an alert is provided to an appropriate party in a step 244.

If a server can be contacted, as ascertained in scan 236, the date of the last modification of the prohibition list is obtained from that server in a step 246. In a comparison 248, the processing unit 30 then determines whether the prohibition list has been modified since the list was cached by the processing unit. Where such a modification has occurred, a copy of the prohibition list is obtained in a step 250. The encryption status of the obtained list is investigated at 252. If the copy of the prohibition list. Finding a nonencrypted copy of the

prohibition list leads to an alert status in a step 254, while an encrypted packet is investigated at 256 to determine with the packet was sent under the system user account. A properly sent packet results in an update of the cached copy of the prohibition list in a step 258 and a notification of the update to all servers in a step 260.

There are two primary methods for submitting or collecting data via the Uni-Directional Data Submission and Collection Protocol ("UDSCP"). Pursuant to the first method, submissions can be directed to either a primary server 14 or a secondary server 22. All submissions are then collected at a central server, where the submissions are processed by an application-specific server-side module. This method would be particularly useful for collecting all form submissions on one server where they can be incorporated into a LAN-based mail system. In the second method, a submission can be again directed at either a primary server 14 or a secondary server 22. The submissions are collected on the servers to which they were originally submitted (or are shunted using the standard load collection rules). The submissions are then processed by an application-specific server-side module. This module could, for example, e-mail all of the submissions to an Internet e-mail address.

Fig. 9 illustrates program steps undertaken by digital processing circuits of processing unit 30 which are functionally modified in in accordance with the UDSCP protocol. These circuit handle data submissions transmitted from user computers 12, particularly from unidirectional data submission and collection circuit 90 of processing unit 54, and from other servers. In a first inquiry 262, the processing unit 30 inquires whether an incoming packet is a data submission. If so, another inquiry 264 is made as to whether the data has to be collected immediately. If immediate collection is called for, the next question 266 entertained by the processing unit 30 is whether the data has to be collected at the source server. If not, the packet is passed in a step 268 through to a module that handles the final data collection. This

module handles e-mailing the submission, recording it in a database, writing it to a file, or preforming any other necessary server-side task with the data. Subsequently, in an investigation 270, it is checked whether the request has been processed by the data collection module. If so, the packet is removed in a step 272 from the UDSCP submission processing queue, assuming that is where the packet was obtained. Then a status report is issued at 274 indicating successful data packet transmission. If investigation 270 reveals that the request has not been processed by the data collection module, the processing unit 30 questions at 276 whether the failure to process was due to a possibly temporary condition. If the answer to this question 276 is negative, a status report 278 is issued describing the error condition. If the answer to the question 276 is affirmative, a status report 280 is issued describing the condition and indicating the possibility of delay in processing the data. The packet is then added in a step 282 to the UDSCP processing queue.

If an incoming packet is a data submission which does not have to be collected immediately, as determined at inquiries 262 and 264, a status report 284 is issued indicating success and the packet is then added in a step 286 to the UDSCP processing queue. If an incoming data packet is a data submission which has to be collected immediately at the source server, as determined at inquiries 262 and 264 and in response to question 266, a check 288 is made as to whether the host server is the source server. If so, the packet is passed in step 268 through to a module that handles the final data collection. If not, processing unit 30 conducts an investigation 290 as to whether the source server can be contacted. If no contact is possible, a status report 292 is generated indicating that there will be a delay in processing the data and the packet is then added to the UDSCP processing queue in step 286. If the source server can be contacted, the data is transmitted to that server in a step 294. If the UDSCP function-modified generic processing circuits of processing unit 30 are provided with a packet

other than a data submission, a report is produced in a step 296 indicated that the packet is of unknown type.

Fig. 10 illustrates steps in a maintenance loop undertaken by generic digital processing circuits in processing unit 30 which are functionally modified in accordance with the UDSCP protocol. First, an inquiry 298 is made as to whether there are entries in the UDSCP submission processing queue. If so, the first entry is read in a step 300. Subsequently, processing unit 30 decides at junction 302 whether more than X seconds have passed since a date and time stamp on the entry. If not, the UDSCP submission processing queue is investigated at 304 to ascertain whether any further entries are in the queue. If so, the next entry is read in a step 306 and again the processing unit 30 decides at junction 302 whether more than X seconds have passed since a date and time stamp on the entry. If the time passed is greater than that predetermined limit, then a query 308 is made as to whether the server is too busy to handle the data submission. If the server is indeed too busy, processing unit 30 questions at 310 whether the data has the be collected immediately. If the data collection is not urgent, processing unit 30 determines at 312 whether the date and time stamp on the entry was earlier than a particular time. If the date and time stamp is recent, processing unit 30 returns to investigation 304 to check any remaining data submissions in the UDSCP submission queue.

If the server is not too busy to handle a data submission (query 308), if the data has to be collected immediately (question 310), or if data and time stamp indicates a certain age to the data submission (determination 312), processing unit 30 determines at a decision junction 314 whether the data has to be collected by the source server. If not, the packet is passed in a step 316 to a module that handles the final data collection. This module handles e-mailing the submission, recording it in a database, writing it to a file, or preforming any other necessary

server-side task with the data. Subsequently, processing unit 30 checks at 318 whether the data submission was processed by the data collection module. If processing has indeed occurred, the packet is removed from the UDSCP submission processing queue in a step 320 and the processor 30 returns to investigation 304 to ascertain whether any further entries are in the queue. If the data submission packet has not been processed, which is discovered at 318, an inquiry 322 is made as to whether the failure to process the request is due to a possibly temporary condition. If not, a notification 324 is generated for alerting an appropriate person as to the failure. If so, the date and time stamp on the data submission is updated in a step 326.

If processing unit 30 determines at decision junction 314 that the data has to be collected by the source server and further determines at a subsequent decision junction 328 that the host (itself) is the source server, then the packet is processed (step 316). Alternatively, if the source server must do the data collection and is a different computer, as determined at decision junction 328, then an attempt 330 is made to contact the source server. If the source server cannot be contacted, the date and time stamp on the data submission is updated in step 326. If the source server is available, the data submission is transmitted to the source server in a step 332 and the packet is removed from the UDSCP processing queue in a step 334.

As discussed above, the MCMP protocol handles Load Distribution, User Authentication, and Encryption. All other functions are handled by sub-protocols. The MCMP protocol has four sub-protocols. These are the Code Module Exchange Protocol (CMXP), the Uni-directional Data Submission and Collection Protocol (UDSCP), the Metered Delivery and Billing Protocol (MDBP), and the Distributed Processing Protocol (DPP). This set of protocols can be expanded in the future to add additional functionality to the MCMP protocol.

Basic MCMP Packet Structure

All MCMP packets consist of a MCMP header, followed optionally by one or more resource identifiers, and a data area called the *packet body* (Fig 1). The entire size of an MCMP packet can be calculated as 128+RsrcIDLen+PacketSize, where RsrcIDLen and PacketSize are elements of MCMP header (see below).

The MCMP header identifies the sub-protocol to which the packet belongs, as well as the type of packet. In addition, the MCMP header contains load distribution, user authentication, and encryption information. The Resource identifiers identify any resource or resources referred to in the packet (the ResourceReq flag being set). The MCMP packet body contains packet-type specific information and is always interpreted by a subprotocol handle. The packet body is optional and can be omitted b ysetting the PacketSize element of the MCMP headet to be 0.

The MCMP header structure is defined as:

MCMPHeader,128		$$ MCMP header structure
·	MPVersion,4	$$ Master protocol version
·	ProtoVendor, 8	$$ Protocol vendor
·	ProtoID, 8	$$ Protocol ID
·	ProtoVer, 4	$$ Protocol version
·	TransID, 2	$$ Client Assigned Transaction-ID
·	PacketType, 2	$$ Protocol-specific Packet Type
·	PacketVersion, 4	$$ Packet version number
·	PacketSize, 4	$$ Size, in bytes, of packet body
·	OrigIP, 4	$$ Originating Host IP Address
·	OrigPort, 2	$$ originating Host Port Number

·	UserID,10	$$ User ID for client authentication
·	Password, 10 -	$$ Password for client authentication
·	RsrcIDLen,2	$$ Resource identifier length
·	RsrcIDs,2	$$ Number of resource identifiers
·	RsrcSrcIP,4	$$ Resource source IP
·	Flags, 2	$$ Flags; see functions below
·	, 64	$$ Reserved

* Flags:

Shunted = bit (Flags,l)	$$ Packet has been shunted
Shuntable = bit (Flags,2)	$$ Packet can be shunted
Encrypted = bit (Flags,3)	$$ Packet was encased in an encryption packet
ResourceReq = bit (Flags,4)	$$ Packet constitutes a request for a resource

The elements of this structure are described in more detail below.

MPVersion: This is a 4-character version identifier for the Master Protocol. If the structure of the MCMP header or any other major component of the Master Protocol structure is revised, this version number is incremented.

ProtoVendor: This is an 8-byte text literal that describes the software vendor initial responsible for maintaining the sub-protocol specificiation for the sub-protocol to which the packet belongs.

ProtoID: This is an 8-byte text literal assigned by the software vendor specified in the ProtoVendor element. This identifies the sub-protocol to which the packet belongs. The combination of ProtoVendor and ProtoID uniquely identifies a sub-protocol.

ProtoVer: This is a 4-byte text string that specifies the version of the sub-protocol specified in the ProtoVendor and ProtoID elements. The first to characters are the major version, the second two the minor version. All characters must be used, so if the major version is one character long, it must be written as 01, not 1. This value does not contain a decimal point. For example, version 2.4 would be 0 2 4 0.

Packet-Type Identifier (PacketType): A two-byte integer assigned by the vendor that uniquely identifies the packet type within a particular sub-protocol.

PacketVersion: This is a 4-character identifier for the packet version. When the structure of a packet is changed, this version number is incremented. This allows a sub-protocol handler running on an MCMP server to deal with both old and new packet structures, should a packet structure need to be altered. The format for the string is the same as the format of the ProtoVer element of the MCMP header.

Shunted flag (Shunted): A flag indicating whether this packet has been shunted as the result of a service hand off.

Shuntable flag (Shuntable): A flag indicating whether this packet can be shunted. This flag and the Shunted flag are mutually exclusive.

Encrypted flag (Encrypted): A flag indicating whether the packet was encrypted. This flag is cleared when a packet is placed in the MCMP Server incoming packets queue, unless the packet is place there by the decryption system, in which case the flag is set.

Request-Resource flag (ResourceReq): Indicates whether the packet constitutes a request for a resource.

Number of Resource Identifiers (RsrcIDs): Dpecifies the number of resource identifiers following the MCMP header structure.

Resource Identifier Length (RsrcIdLen): Specifies the combined length of all resource

identifiers following the MCMP Header structure.

Originating Host Address (OrigIP, OrigPort): This is the IP address of the host from which the request originated. If the packet was shunted, this is the IP address of the host that originally transmitted the request, not the address of the server that forwarded the request.

Resource Source IP (RsrcSourceIP): This is the IP address of the host on which the original copy of the requested resource resides, if the Request-Resource flag is true.

Cached Copy Date/Time Stamp (CacheDate, CacheTime): This is the date and time stamp of the cached copy of the resource, or zero if no cached copy exists. If the date and time stamp of the resource match this date and time stamp, the resource content will not be returned.

Size of packet body (PacketSize): The size (in bytes) of the packet body following the MCMP Header and (if present) the resource identifiers.

User ID and Password for client authentication (UserID, Password): A user ID and Password used to authenticate the client. The authority for a user's ID and Password is the Resource Source IP server. If a request is made to a secondary server for a password-protected resource, the secondary server must check with the primary (or source server) for password authentication. This information can be cached for the duration of the session.

Transaction ID (TransID): A unique ID assigned by the host initiating the transaction. This is used to track a transaction over one or more service hand-offs. For an encryption packet, this should be set to zero (although it can be set to a non-zero value in the encased packet).

MCMP Resource Identifiers

If the packet refers to one or more resources (the ResourceReq flag is set), the

Resource identifiers identify the resource or resources to which the packet refers. Resource identifiers are null-terminated text strings. The basic format for a resource identifier is:

- type:[locator/s;]length[;date[,time]]

Where:

type	Identifies the resource type; the possible values for this argument are sub-protocol-specific.
locator/s	One or more values (in double-quotes if they contain commas; double up the double-quotes where they are used within the value) used to locate the resource.
length	The amount of the resource to read (in units specific to the sub-protocol). This is always the last item in the resource identifier. It can be in the format "*n*" to specify n units starting at the beginning of the resource, the format "*n*:*n*" to specify a range of units (inclusive), the format "*n*,*n*" to specify a starting unit and number of units, or "*" to specify the entire resource.
date	The date, in the format mm/dd/yyyy that the cached copy of the resource was last modified. This field accepts both single and double digits for the month and day, although the year must be specified as a full 4-character string. Forward-slashes must be used as separators.
Time	The time, in the 24-hour format hh:mm:ss, that the cached copy of the resource was last modified. hh may be a single or double digit number, and mm and as must be double digit numbers (use a leading zero if necessary).

NSDOCID: <WO 9907007A2 I >

The Resource ID is optional, and does not need to be included in a MCMP packet, so long as the ResourceReq flag is not set.

Some example Resource ID's are:

data:"\catharon\demos\media., "video.bin", "play30";*

prohibitedlist:*

dir:"\";*

data:"\catharon\demos\","mag2.dat";0/256

data:"\training","air.bin.","Rudder";*;10/03/1995,4:03:30

MCMP Packet Types

The following subprotocols are not sub-protocl specific and are accordingly defined for the MCMP protocol:

MCMP_Encrypt	= h0002	$$ Encryption
MCMP_Status	= h0003	$$ Status Report
MCMP_PerfStatReq	= h0006	$$ Performance Statistics Request
MCMP_PerfStatResp	= h0007	$$ Performance Statistics Response
MCMP_UserIReq	= h0008	$$ User Information Request
MCMP_UserIResp	= h0009	$$ User Information Response
MCMP_EchoReq	= h000a	$$ Echo Request
MCMP_EchoResp	= h000b	$$ Echo Response

These packet types are described in detail below.

ENCRYPTION (MCMP_Encrypt): The encryption packet is used when data must be encrypted. A packet to be encrypted is encased in the data area of a MCMP System encryption packet. The MCMP Header for the encryption packet is:

PacketType:	MCMP_Encrypt (hO002)
PacketSize:	Size of encased packet in encrypted form plus 32 bytes
ResourceID's:	None
Shuntable:	Inherited from encased packet
ResourceReq:	False

The packet body of the encryption packet is:

Bytes 0-31	Encryption header
Bytes 32-end	Encrypted packet

The header for the encryption packet is:

PT_Encrypt_Header,32

- DecryptLess,8 $$ Code module to handle decryption
- DecryptUnit,8
- EncodingMethod,2 $$ Encoding method
- ,14 $$ Reserved

STATUS REPORT (MCMP_Status): The status report packet is used to return the status of an operation. When used, it is always a response to a previously transmitted request. The status packet contains detailed error information, including an English language description of the error or status value which can be displayed by the application if it cannot interpret the error code.

A status report packet body consists of one or more information fields, which can vary in length. The first information field is always a Status Report Header. Each information field consists of an 8-byte header which indicates the length and type of the information field, followed by the field itself.

The MCMP Header for the status report packet is:

PacketType: MCMP_Status (hO002)

PacketSize: Variable

ResourceID's: None

Shuntable: No

ResourceReq: No

The Information Field Header for the status report packet is:

```
MCMP_StRprt IFldHdr,8
    · IfldSize,2        $$ Size of information field
    · IfldClass,1       $$ Field alass (1=Standard, 2=Protocol Specific)
    · IfldType,2        $$ Field type
    · , 3               $$ Reserved
```

Following are the standard information field types (where IFldClass=l):

```
MSR_Header          = 1     $$ Header
MSR_ShortDesc -     = 2     $$ Short Description
MSR LongDesc        = 3     $$ Long Description
MSR_DetailDesc      = 4     $$ Detailed Description (Technical)
MSR_XErr70          = 102   $$ LAS 7.0 Execution Error Data
MSR_XErr70do        = 103   $$ LAS 7.0 Execution Error-do-stack
```

These information field types are described in detail below:

The generic status report header (MSR_Header) is always present in all status report packets, and is always the very first information field in the packet. It has the following structure:

```
MSR Header_Struc,32
    ProtoVendor, 8      $$ The Vendor of the protocol reporting the error
```

- ProtoID, 8 $$ The ID of the protocol reporting the error
- ProtoVer, 4 $$ The version of the protocol reporting the error
- Severity, 1 $$ severity of error:
- * -1 = Notification of success
- * 0 = Warning (operation will proceed anyway, but there may be a problem)
- * 1 = Error (operation cannot proceed)
- * 2 = Unexpected error
- ProtoSpecific, 1 $$ Protocol-specific error-flag.
- ErrorType, 2 $$ Sub-Protocol-Specific error type
- ErrorCode , 2 $$ Sub-Protocol -Specific error code
- , 6 $$ Reserved

If the ProtoSpecific flag is set, then ErrorType and ErrorCode are protocol-specific. Otherwise, ErrorType is one of the following:

ERRT _ Zreturn = 1 $$ *zreturn* error

ERRT_XErr = 2 $$ TenCORE Execution error

ERRT_CErr = 3 $$ TenCORE Condense error

ERRT_Dosdata = 4 $$ Catharon dosdata-style error

For anything in the MCMP_Status packet that is protocol specific, the ProtoVendor and ProtoID from the Status Report Header are used to identify the protocol.

The Short Description information field type (MSR_ShortDesc) is a short description of the error, 40 characters long or shorter, that can be used in a list or wherever a brief, friendly error description is needed. This packet is 40 bytes long, and is structured as follows:

MSR_ShortDesc strtlc, 40

ShortErrDesc,40 $$ Short description of error

The Long Description information field type (MSR_LongDesc) is a longer description of the error, which can vary in length up to 2048 characters. This description can span multiple lines, with each line terminated by a carriage return (h0d). The length of this description is determined by the length of the information field, and the entire content of the information field is one long buffer variable containing the description as text. There is no maximum length to a line, and lines may be word-wrapped at any position when this description is displayed.

The Detailed Description information field type (MSR_DetailDesc) is a detailed technical description of the error, with all diagnostic error information included. For example, this might be a standard TenCORE execution error as it would be written to the catharon.err log file by the Catharon error handler. This can vary in length, up to 4096 characters. The description can span multiple lines, with each line terminated by a carriage return (h0d). Lines must be no longer than 80 characters. Lines longer than 80 characters may be truncated at display time. This description is never word-wrapped, and is always displayed in a fixed pitch font, allowing items on separate lines to be aligned and formatted using spaces (tables could be created using this method). The length of this description is determined by the length of the information field, and the entire content of the information field is one long buffer variable containing the description as text.

The TenCORE 7.0 Execution Error Data (MSR_XErr70) is an exact snapshot of the data generated by a TenCORE execution error, and returned by TenCORE in the execution error memory block. It is 256 bytes long. This information field type is normally only included if the error being reported is a TenCORE execution error.

The TenCORE 7.0 Execution Error To- Stack (MSR_XErr70do) is an exact snapshot of the TenCORE execution error -do- stack. The size of the data varies based on the size of the TenCORE -do- stack at the time of the error.

PERFORMANCE STATISTICS REQUEST (MCMP_PerfStatReq): The performance statistics request packet requests a server's current performance and load statistics.

The MCMP Header of a performance statistics request is:

PacketType:	MCMP_PerfStatReq (h0005)
PacketSize:	0
ResourceID's:	None
Shuntable:	No (because this is a request for the statistics for the server to which it is addressed, shunting me packet would cause meaningless results)

The response to this packet should be either an MCMP_PerfStatResp or a MCMP_Status packet.

PERFORMANCE STATISTICS REPORT (MCMP_PerfStatResp): This packet is a response to an MCMP_PerfStatReq packet and contains a performance statistics report for the server to which it was addressed.

The MCMP Header of a performance statistics report is:

PacketType:	MCMP_PerfStatReq (h0006)
PacketSize:	32
ResourceID's:	None
Shuntable:	No
ResourceReq:	No

The Packet Body of a performance statistics report is:

PerfStats,32

- Pusage,1 $$ Processor usage (percent)
- CurReqs,2 $$ Current number of requests being processed
- TotalRegs,2 $$ Total number of requests the can be processed
- ShuntRegs,2 $$ Threshold, in requests, at which shunting begins
- PmemTotal,4 $$ Total physical memory on system
- PmemUsed,4 $$ Used memory on system
- VmemTotal,4 $$ Total virtual memory on system
- VmemUsed,4 $$ Used virtual memory on system
- AreqPMethod,1 $$ Current method for processing additional requests
- ,8 $$ Reserved

The elements of the packet body structure are, in detail:

PUsage:Current processor usage percentage (0% to 100%). Set to -1 if processor usage percentage is not available.

CurReqs: Approximate number of requests currently being processed.

TotalReqs: Total number of requests that can be processed at one time.

ShuntReqs: Maximum number of requests before shunting occurs. This is usually less than TotalReqsto allow some extra system resources for the purpose of shunting requests.

PmemTotal: Number of bytes of physical memory on server, or -1 if amount not known.

PnemUsed: Number of bytes of physical memory that have been used, or -1 if amount not known.

VmemTotal: Number of bytes of virtual memory available on server, -1 if not known.

VmemUsed: Number of bytes of virtual memory in use, or -1 if not known.

AreqPMethod: Method that will be used by the server deal with new incoming requests, based on the other statistics in this packet. This can have the numerical value 1, 2, or 3. 1 indicates that new requests will be processed normally; 2 indicates that new requests will be shunted; 3 indicates that new requests will be refused.

USER AUTHENTICATION INFORMATION REQUEST (MCMP_UserIReq): This packet requests user authentication information on a particular user. This packet must be encrypted, and is sent only from a secondary server to a primary server. The receiving server must check that the sender is listed as a secondary server before responding to this request.

The expected response is either an MCMP_Status packet or an MCMP_UserIResp packet.

This packet uses a special type of resource identifier, which is defined as follows:

type:user[,uadbitem];length[;date,time]

Where:

type	Identifies the resource type; is always "useradb"
user	The name of the user in question
uadbitem	Path to user authentication database item to retrieve. If omitted, a tree-file is resumed containing the entire user authentication data tree for the specified user. The root of the resumed tree-file is equivalent to \LocalUsers\username in the user database file.
length	Portion of item to be resumed.

Examples:

useradb:JohnS.\Catharon\RAdmin\Rights:*:03/13/1995,12:20:48

useradb:HugoC;*

useradb:JDouglas,\XYZWidSt\WDBP\GroupMembership;256;S/12/1996, 01:00:30

The User Authentication Database (UADB) is stored in a tree-file. User information is stored in \LocalUsers. Inside the \LocalUsers folder are folders for each user, named based on the user's ID. In each user's folder are folders for each vendor (i.e. "Catharon", "CTC", etc.), and inside the vendor folders are folders for each protocol defined by that vendor. The contents of a protocol folder are protocol-specific. The path specified in the resource ID is rooted at \LocalUsers\username.

Basic user authentication information is stored in \Catharon\MCMP\BaseAuthData. This is structured as follows:

UserAuthData,32

- UserID,10 $$ Userts name/ID
- UserPass,10 $$ User's password
- ExpTime,3 $$ Time, in seconds, before data
- $$ expires (10 to 864000).
- BinUID,4 $$ Binary User ID
- ,5 $$ Reserved

If a secondary server sends a request in the form:

useradb:usez,2ame;*[;date,time]

the entire user authentication tree is retrieved for the specified user. The ability to read a specific item from the user authentication tree is provided for future use and expandability.

After retrieving user authentication data, that data can be cached for the period of time specified in the ExpTime element of the UserAuthData structure. The user authentication data may not be cached longer than the specified time.

The MCMP Header of the User Authentication Information Request is:

PacketType:	MCMP_UserIReq (h0008)
PacketSize:	0
ResourceID's:	One; specifying the user to retrieve information about, and what information to retrieve.
Shuntable:	No (must be processed by primary server because the primary server is the only authoritative source of information on the user).
ResourceReq:	Yes

USER AUTHENTICATION INFORMATION RESPONSE (MCMP_UserIResp): The response to an MCMP_UserIReq packet, this packet contains the requested information in its data area. This data is either the raw data read from the requested data block in the user database, or (if uadbitem is omitted) is a tree file containing the entire user information tree for the specified user, with the root of the file being equivalent to \LocalUsers\username in the UADB.

The MCMP Header of the User Authentication Information Response is:

PacketType:	MCMP_UserIResp (hO009)
PacketSize:	Variable
ResourceID's:	None
Shuntable:	No
ResourceReq:	No

ECHO REQUEST (MCMP_EchoReq): This packet is used to time the connection to a particular MCMP host. When this packet is received by an MCMP host, a MCMlP_EchoResp packet is immediately sent back. The data area can contain any data, up to a maximum size of

2048 bytes. The return packet's data area will contain the same data.

The MCMP Header of the Echo Request is:

PacketType:	MCMP_EchoReq (hOOOa)
PacketSize:	Any
ResourceID's:	None
Shuntable:	No
ResourceReq:	No

ECHO RESPONSE (MCMP_EchoResp): This packet is sent in response to an MCMP_EchoReq packet.

The MCMP Header of the Echo Response is:

PacketType:	MCMP_EchoResp (hOOOb)
PacketSize:	Same as original MCMP_EchoReq packet
ResourceID's:	None
Shuntable:	No
ResourceReq:	No

Some files in a directory may support additional access pennission information. For example, a tree file could contain information on access permission for individual units within the tree file.

CMXP Resource Identifiers

Resource Identifiers for CMXP packets are defined as follows:

type:patht,file[,unit]];length[;date,time]

Where:

type Identifies the resource type. This can be either "Data" to read data from the specified resource, or "Dir" to read a directory of contained

resources.

path Path to a directory...must be at least a backslash "\". If file and unit are not specified, the directory is considered the resource to be read; otherwise, the file or unit referenced is assumed to be located in the specified directory. If file and unit are not specified, then type must be "Dir".

file A filename. If unit is not specified, the file is considered the resource; otherwise, the unit is assumed to be located in the specified file. A file resource can be accessed with both the "Dir" and "Data" resource types; "Dir" will reference the list of contained units, while "Data" will reference the actual data contained in the file.

unit A unit name. If specified, the unit is considered to be the resource. This can be accessed with both the "Dir" and "Data" resource types; "Dir" will reference the list of sub-units, while "Data" will access the data contained in the unit.

length The portion of the resource to read. If typeis "Data", this value is in bytes. If type is "Dir", this value is in directory entries.

date/time Can only be specified in a request packet. Causes the recipient process to ignore the request unless the resource has been modified since the date/time specified. This can be used in conjunction with the CMXP_ReadReq packet to request that a resource be sent only if it has changed since it was cached on the client.

CMXP Packet Types

The following is a list of the packet types used by the CMXP protocol at this time. The

functionality of the CMXP protocol can be expanded in future by adding to this list of packet types.

CMXP_ReadRsrcReq = h0002

CMXP_ReadRsrcResp= h0003

CMXP_WriteRsrc = h0004

CMXP_CreateRsrc = h0005

CMXP_DestroyRsrc = h0006

CMXP_RenameRsrc = h0007

CMXP_CopyRsrc = h0008

CMXP_MoveRsrc = h0009

CMXP_AltSListReq = h000a

CMXP_AltSListResp = h000b

These packet types are described in detail below.

READ RESOURCE REQUEST (CmxP_ReadRsrcReq): This packet is a request to read one or more resources. It is sent from a client to a server to download a resource, sent from a client to a client to request a code module transfer for a plug-in module, or sent from a server to a server to request transfer of the appropriate resource to service a client request. A CMXP_ReadRsrcReq packet can request either resource content, resource information, or both. Because the definition of a resource includes file directory and code module directories, this packet can also be used to request a list of files in a directory or code modules in a file.

This packet is responded to with a series of packets (one for each request resource). These packets are either CMXP_ReadRsrcResp (if the resource was successfully read) or MCMP_Status (if there was an error reading the resource).

The MCMP Header of a read resource request packet is:

PacketType: CMXP_ReadRsrcReq (honey)

PacketSize: 32

ResourceID's: One or more, Identifying resources to read

Shuntable: Yes

ResourceReq: Yes

The Packet Body of a read resource request packet is:

```
ReadRsrcReqHeader,32
    RHFlags,2          $$ Flags
    ,30                $$ Reserved
    * Flags:
    IncludeInfo = bit(RHFlags,1)      $$ Include resource information
    IncludeData = bit(RHFlags,2)      $$ Include resource content
    IdlePreCache = bit(RHFlags,3)     $$ Request is the result of an idle-time
                                          pre-caching operation.
```

The elements of the packet body are, in detail:

IncludeInfo: If set, this flag causes infonnation about the resource to be returned.

IncludeData: If set, this flag causes the resource content to be returned.

IdlePreCache: If set, indicates the the request is the result of an idle-time pre-caching operation initiated by the client without user involvement. A CMXP server processes packets with this flag clear before packets with this flag set are processed. When the load on a CMXP server becomes too high for it to deal with all requests, and it can not shunt requests, requests with this flag set will be dropped before requests with this flag clear.

IncludeInfo and IncludeData can both be set in the same request In this case, the response is the resource information followed by the resource content. This is the most

common request type. At least one of these flags must be set in each request packet

READ RESOURCE RESPONSE (CMXP_ ReadRsrcResp): Sent in response to a CMXP_ReadRsrcReq packet, this packet contains the requested information. Note that this packet is never sent in response to a CMXP_ReadRsrcReq if an error condition exists; instead, a MCMP_Status packet is sent.

Depending on the state of the IncludeInfo and IncludeData flags in the CMXP_ReadRsrcReq packet, the packet body may contain resource information and/or resource content. The resource information, if it is present, always comes first in the packet body, followed by the resource content, if present. The size of the resource information can be determined by reading the ResourceInfoSize element of the version of a program can be sucessfully merged with code modules from an old version of the program.

RsMaxSubs: The maximum number of subsidiaries that the resource can contain.

RaSizeAInfo: The size, in bytes (1 to 8), of the associated information for the resource (RsAInfo).

RsSizeSubName: The maximum length, in characters, of the name of a subsidiary of the resource.

RsHeight, RsWidth: The height and width of the bounding rectangle of the resource at its default scaling, if applicable. This is used for object-based drawings, images, etc.

RsPTime: The playing time, in seconds at the default playing speed, of the resource (if appliable); used for video clips, wav files, midi files, animations, etc.

WRITE RESOURCE (CMXP_WriterSrc): This packet writes data to the specified resource. This requires sending a packet to the primary server that cannot be shunted, so this can increase server load. For form submissions and other uni-directional submissions, the UDSCP protocol is recommended over the write functions of the CMXP protocol.

A CMXP_WriteRsrc request may fail due to, among other things, the fact that the user is not allowed to write to the specified resource. This situation may be remediable by repeating the request with a user-id and password specified (in this case, the request must be encased in a MCMP_Encrypt packet).

There are two variations of the status code returned when access is denied due to failed user authentication. One variation indicates that the client should prompt for a user name and password. This is a "hint" to the client that the resource may be accessible via a user name and password. It is not conclusive. In other words, the variation on the Access Denied code does NOT indicate whether access is really available to anyone; i just indicates whether the client should ask. There may, for example, be a resource designed to be accessed under program control, and not under user control, which requires user authentication of an "automation user", and which denies access the rest of the time without even prompting for a user-id/password.

The packet body contains the data to be written to the resource.

The MCMP Headerof the write resource packet is:

PacketType:	CMXP_WriteRsrc (h0004)
PacketSize:	Variable
ResourceID's:	One; the ID of the resource to-wnte to
Shuntable:	No
ResourceReq:	Yes

CREATE RESOURCE (CMXP_CreateRsrc): This packet creates a subsidiary in the specified resource. This includes files, directories, and units. The same rules regarding user authentication apply to this packet as apply to the CMXP_WriteRsrc packet.

The MCMP Header of the create resource packet is:

BNSDOCID: <WO__9907007A2_I_>

PacketType: CMXP_CreateRsrc (h0005)

PacketSize: Variable

ResourceID's: One; the ID of the resource In which to create the new subsidiary.

Shuntable: No

ResourceReq: Yes

The Packet Body of the create resource packet is:

NewRsrc,300

- RsSize,4 $$ Size, in bytes, of resource
- RsAInfo,8 $$ Associated information (if applicable)
- RsMaxSubs,2 $$ Maximum number of subsidiaries
- RsSizeAInfo,1 $$ Size, in bytes, of associated information
- RsSizeSubName,2 $$ Maximum length of a subsidiary's name
- RsName,256 $$ Resource Name
- RsName2,8 $$ Secondary Resource Name
- RsType,8 $$ Resource type
- ,11 $$ Reserved

The elements of the packet body are, in detail:

ReSise: The size, in bytes, of the new resource. If creating a TenCORE nameset or dataset, this must be a multiple of 256 bytes, and is equivalent to the *records* argumen of the -createn- commnand multiplied by 256 (to convert from records to bytes). When creating new directories, this is ignored.

R-AInfo: Associated information for the resource, if applicable (see CMXP _CreateRsrc above)

RsMaxSubs: The number of subsidiaries allowed in the resource, if applicable. This is required for TenCORE namesets, and is equivalent to the names argument of the -createn- command. In most cases, when not dealing with TenCORE namesets, this is ignored.

R-SizeAInfo: The size of the resource's associated information. For namesets, this is equivalent to the infolength argument of the -createn- command.

RsSizeSubName: Maximum length of a subsidiary's name. For TenCORE namesets, this is equivalent to the namelength argument of the -createn- command. In most cases when-not dealing with TenCORE namesets, this is ignored.

ReName: The name of the resource to create. The length of this name and the rules for allowed characters depend on the type of resource being created.

RaName2: This is a secondary name for the resource. It is not currently used, but is provided for future use. This may be used, for example, to specify a short file name alias to go with a long filename in Windows 95/NT.

RaType: This specifies the type of resource being created. Note that not all resource types are necessarily valid for all possible resources in which they could be created (i.e., one cannot create a file inside of a unit). Where only one resource type is possible for a particular containing resource, the value 'default' is used (for example, the only type of resource that can be created inside a TenCORE nameset is a block). This value is an 8-byte text literal.

The following resource types are currently defined for the CMXP protocol:

course	A course file (TenCORE nameset with .CRS extension)
group	A group file (TenCORE nameset with .GRP extension)
nameset	A gentral purpose nameset file (TenCORE nameset with .NAM extension)
roster	A roster file (TenCORE nameset with .RTR extension)

source	A source file (TenCORE nameset with .SRC extension)
studata	A student data file (TenCORE nameset with .SDF extension)
tpr	A Producer file (TenCORE nameset with .TPR extension)
binary	A binary file (TenCORE nameset with .BIN extension)
file	An operating system file
dir	An operating system folder or directory
dataset	A TenCORE dataset
tree	Catharon tree-file
default	The default type for the container
block	A data block (in a nameset or tree-file)
folder	A folder (in a time-file)

DESTROY RESOURCE (CMXP_DestroyRsrc): This packet destroys the specified resource. The same rules regarding user authentication apply to this packet as apply to the CMXP_WriteRsrc packet. The MCMP Header of this packet is:

PacketType:	CMXP_DestroyRsrc (h0006)
PacketSize:	0
ResourceID's:	One; the ID of the resource to destroy
Shuntable:	No
ResourceReq:	Yes

RENAME RESOURCE (CMXP_RenameRsrc): This packet renames the specified resource. The same rules regarding user authentication apply to this packet as apply to the CMXP_WriteRsrc packet. The packet body contains the new name for the resource. The MCMP Header of this packet is:

PacketType:	CMXP_RenameRsrc (hO007)

PacketSize: 272

ResourceID's: One; the ID of the resource to rename

Shuntable: No

ResourceReq: Yes

The Packet Body of the rename resource packet is:

RenameRsrc,272

- ResourceName,256 $$ New name for resource
- ResourceSecondaryName,8 $$ Secondary name for resource (if applicable)
- ,9 $$ Reserved

COPY RESOURCE (CMXP_CopyRsrc): This packet copies the specified resource. The same rules regarding user authentication apply to this packet as apply to the CMXP_WriteRsrc packet. The MCMP Header of the copy resource packet is:

PacketType: CMXP_CopyRsrc (h0008)

PacketSize: 0

ResourceID's: Two; the first is the location of the resource to copy, the second the location to create the new copy of the resource.

Shuntable: No

ResourceReq: Yes

MOVE RESOURCE (CMXP_MoveRsrc): This packet moves the specified resource. The same rules regarding user authentication apply to this packet as apply to the CMXP_WriteRsrc packet. The MCMP Header of the move resource packet is:

PacketType: CMXP_MoveRsrc (h0009)

PacketStze: 0

ResourceID's: Two; the first ts the old location of the resource, the second the new location for the resource.

Shuntabte: No

ResourceReq: Yes

ALTERNATE SERVER LIST REQUEST (CMXP_AltSListReq): This packet requests a list of secondary servers available for the specified resource. The server should respond with an CMXP_ AltSListResp packet, except in the case of an error, in which case an MCMP_Status packet should be returned. The MCMP Header of this packet is:

PacketType: CMXP_AltSUstReq (h000a)

PacketSize: 0

ResourceID's: One - The resource for which to list secondary servers.

Shuntable: Yes

ResourceReq: Yes

ALTERNATE SERVER LIST RESPONSE (CMXP_AltSListResp): This packet is sent in response to an CMXP_ AltSListReq packet and contains the list of alternate servers. The MCMP Header is:

PacketType: CMXP_AltSUstResp (h000b)

PacketSize: Variable

ResourceID's: None

Shuntable: No

ResourceReq: No

The Packet Bodyof the CMXP_AltSListResp packet is:

AltSList(nn),16

·	IP,4	$$ IP Address of server
·	Port, 2	$$ Port on server to access
·	Load, 1	$$ Last known load on server
·	Ping,4	$$ Last known ping-time to server
·	,4	$$ Reserved

UDSCP Packet Types

The following is a list of the packet types used by the UDSCP protocol at this time. The functionality of the UDSCP protocol can be expanded in future by adding to this list of packet types.

UDSCP_Submission = h0002

UDSCP_QueueStatusReq = h0003

UDSCP_QueueStatusResp = h0004

These packet types are described in detail below:

DATA SUBMISSION (UDSCP_Submission): This is the primary packet type for the UDSCP protocol. It is generated by a client, and then forwarded from server to server until it reaches the collection point. The packet body consists of a UDSCP Header followed by the content of the submission.

The MCMP Header of the data submission packet is:

PacketType:	UDSCP_Submission (hO002)
PacketSize;	32 + Size of data being submitted
ResourceID's:	None
Shuntable:	Yes
ResourceReq:	No

The UDSCP Header is:

UDSCPSubmitHeader,32

- HeaderSize,2 $$ Size of UDSCPSubmitHeader
- DataSize,4 $$ Size of data being submitted
- Cmethod,1 $$ Collection method (1=Central, 2=Secondary)
- CpointIP,4 $$ Collection point IP Address
- Priority, 1 $$ Priority (O=low, 1=normal, 2=urgent)
- Lesson,8 $$ TenCORE Lesson/Unit to process submission
- Unit,8
- Flags, 2 $$ Flags
- ,2

* Flags:

Forwarded = bit(Flags,l) $$ Submission has been forwarded by a server

The elements of the UDSCP header are described in detail below:

HeaderSize: The size, in bytes, of the UDSCPSubmitHeader structure. This value should be read to determine the size of the whole structure, this allowing the structure to be expanded in future without affecting existing code.

DataSize: The size, in bytes, of the content of the submission following the UDSCP Header.

Cmethod: The collection method to be used. This is an integer value. A setting of 1 causes data to be collected and processed at a centeral server, while a setting of 2 allows data to be processed on secondary servers.

CpointIP: The IP address of the collection point (centeral server). This is ignored if CMethod=2.

Priority: The priority of the submission. This is an integer value, and can be either 0 for

low, 1 for normal, or 2 for high priority. UDSCP servers attempt to process high priority submissions immediately, while normal and low priority submissions are held in the UDSCP submission queue and processed when the server would othewise be idle. If a normal or low priority remains in the UDSCP queue for longer than the user configurable time limit, the server will attempt to process it immediately regardless of load or, failing to do so, notify a responsible person. The time limits for low and normal priority submissions are configurable separately, and low priority submissions are usually configured for a longer timeout.

Lesson, Unit: The names of the TenCORE lesson and unit that will process the submission.

Forwarded: This flag is set if the submission has been forwarded by a UDSCP server, or clear if this is the first UDSCP server to deal with the submission (i.e., the submission came from a client).

QUEUE STATUS REQUEST (UDSCP_ QueueStatusReq): This packet requests the status of the UDSCP queue. The expected response is either an MCMP_Status packet or a UDSCP QueueStatusResp packet.

The MCMP Headerof the queue status request packet is:

Packet Type:	UDSCP _ Queue Status Req (h0003)
PacketSize:	0
ResourcelD's:	None
Shuntable:	No
ResourceReq:	No

QUEUE STATUS RESPONSE (UDSCP:QueueStatusResp): This packet is the response to a UDSCP_QueueStatusReq packet. It contains infonnation about the UDSCP server's current UDSCP queue status. The MCMP Header is:

PacketType: UDSCPQueueStatusResp (h0004)

PacketSize: 0

ResourceID's: None

Shuntable: No

ResourceReq: No

The Packet Body of this status response packet is:

QueueStatus,64

- Entries,4 $$ Total number of entries in the queue
- LowEntAge,4 $$ Age of the newest entry in the queue, in seconds
- HighEntAge,4 $$ Age of the oldest entry in the queue
- AvgEntAge,4 $$ Age of the average queue entry
- HighPriEnt,4 $$ Number of high-priority entries in the queue
- LowPriEnt,4 $$ Number of low-priority entries in the queue
- FwdEnt,4 $$ Number of entries which have been forwarded
- ToFwdEnt,4 $$ Number of entries which must be forwarded
- ,32 $$ Reserved

The Metered Delivery and Billing Protocol (MDBP) controls access to pay-for content, including delivering credit to pay for the content, and collecting royalty information after the content has been purchased.

MDBP Packet Types

The MDBP protocol works closely with the CMXP protocol. The CMXP protocol is used to deliver the content in encrypted form. The content is then decrypted when it is

unlocked by the MDBP libraries on the client machine. The content is unlocked when it is paid for with credit on the local machine. Credit can be replenished through the MDBP protocol.

When credit is replenished on the local machine, the royalty information is reported to the credit server, which then handles the appropriate distribution of profits.

In a standard credit-purchasing transaction, three packets are exchanged:

- An MDBP_CreditReq is sent from the client to the server
- The server responds with an MDBP_CreditTransfer
- The client sends an MDBP_PurchaseReport to the server

Before credit can be purchased by a user, that user must be registered with the credit server.

The following is a list of the packet types used by the MDMP protocol at this time. The functionality of the MDBP protocol can be expanded in future by adding to this list of packet types.

MDBP_CreditReq	= 2	$$ Request for additional credit
MDBP_CreditTransfer	= 3	$$ Response to MDBP_CreditReq
MDBP_RegisterUser	= 4	$$ Register a user
MDBP_RegisterUserResp	= 5	$$ Response to MDBP_RegisterUser
MDBP_WriteUserData	= 6	$$ Write user data
MDBP_ReadUserData	= 7	$$ Read user data
MDBP_ReadUserDataResp	= 8	$$ Response to MDBP_ReadUserData
MDBP_PurchaseReport	= 9	$$ Puchasing/Royalty Report

The MDBP packet types are described in detail below.

REQUEST FOR ADDITIONAL CREDIT (MDBP_ CreditReq): This packet requests additional credit from the server. The packet body contains the user's ID code, which is used to

access the user's data in the user database, as well as the amount of credit to be purchased. The user's data includes a billing method to use to pay for the credit. This packet must be encrypted. The expected response is either an MDBP_CreditResp or an MCMP_Status packet.

The MCMP Header of a credit request packet is:

PacketType:	MDBP_CreditReq (h0002)
PacketSize:	22
ResourceID's:	None
Shuntable:	No
ResourceReq:	No

The Packet Bodyof a credit request packet is:

UserID,8	$$ User ID; B-byte integer value
Password,10	S$ User' 8 pasaword
Credit, 4,r	$$ How much credit to purchase (in dollars)

CREDIT TRANSFER (MDBP_CreditTransfer): This packet is the response to an MDBP_CreditReq. It actually performs the transfer of credit from the server to the client. The packet body contains information on the credit transfer. This packet must be encrypted.

The MCMP Header of a credit transfer packet is:

PacketType:	MDBP_CreditReq thOo03)
PacketSize:	20
ResourceID's:	None
Shuntable:	No
ResourceReq:	No

The Packet Body of a credit transfer packet is:

UserID,8	SS The ID of the user who should be receiving this credit

Credit, 4,r $$ The amount of credit purchased (in dollars)

Tserial,8 $$ A serial number used to track the transaction

USER REGISTRATION (MDBP_RegisterUser): This packet is used for the initial registration of a user with a credit server. It causes a new entry to be created in the user registration database. The packet body contains information about the user which will be written into the standard fields in the user's new record. The information can be read and modified at a later time through use of the MDBP_WriteUserData and MDBP_ReadUserData packets. This packet must be encrypted. The expected response to this packet is an MDBP_RegisterUserResp or MCMP_Status packet.

The MCMP Header of a user registration packet is:

PacketType: MDBP_RegisterUser (hO004)

PacketSize:

ResourceID's: None

Shuntable: No

ResourceReq: No

The Packet Bodyof a user registration packet is:

RegisterUser, 512

- Name,54 $$ Full name
- Company, 54 $$ Company name
- Addressl,45 $$ Line 1 of the street address
- Address2,45 $$ Line 2 of the street address
- City, 20 $$ City name
- State,2 $$ 2-letter state abbreviation
- Pcode,16 $$ Postal code (zip code)

·	Country,30	$$ Country name
·	Telephone,16	$$ Telephone number
·	AX,16	$$ FAX number
·	Email,100	$$ E-mail address
·	CardNo,20	$$ Credit card number ('nnnn nnnn nnnn nnnn')
·	CardExpDate,5	$$ Credit card expiration date ('mm/yy' or 'mm-yy')
·	Password, 10	$$ Password
·	,79	$$ Reserved

USER REGISTRATION RESPONSE (MDBP_RegisterUserResp): This packet is the response to an MDBP_RegisterUser packet. It acknowledges the fact that the user has been registered, and returns the user's assigned ID code. The MCMP Header of this packet is:

PacketType:	MDBP_RegisterUserResp (h0005)
PacketSize:	8
ResourceID's:	None
Shuntable:	No
ResourceReq:	No

The Packet Body is:

userID,8	$$ user ID; 8-byte integer value

WRITE USER DATA (MDBP WriteUserData): This packet writes data to the user database. This uses a resource identifier to identify the element in the user registration database to write to. The packet body is the data to be written. This packet uses a special type of resource identifier, which is defined as follows:

type userid[,dbitem];length[;date,time]

Where:

type	Identifies the resource type; is always "mdbpuser"
userid	The ID of the user in question (in hexadecimal)
dbitem	Path to user database item to retrieve. If omitted, a tree-file is resumed containing the entire user data tree for the specified user.
length	Portion of item to be returned.

Examples:

mdbpuser:00000000000003e4,\Catharon\RAdmin\Rights;*;03/13/1995,12:20:48

mdbpuser:0000000000000014;*

mdbpuser:0000000000002aff\XYZWidgt\WDBP\GroupMembership;256;

5/12/1996,01:00:30

The user database contains a system folder and a publishers folder. The system folder contains the data specified in the initial User Registration packet, and can be expanded to contain additional data. The publishers folder contains a subfolder for each publisher, named based on the publisher's name, and the publisher's folder contain, in turn, publication folders, named based on the publication. The organization of data within a publication folder is specific to the publication.

The MCMP Header of the write user data packet is:

PacketType:	MDBP_WriteUserData (h0006)
PacketSize:	Variable
ResourceID's:	1 (The resource to write to)
Shuntable:	No
ResourceReq:	Yes

SNSDOCID: <WO__ 9907007A2.I >

READ USER DATA (MDBP_ReadUserData): This packet reads data from the user database. This uses a resource identifier to identify the element in the user registration database to read from. The format of the resource identifier is the same as that for the resource identifier used in the MDBP_WriteUserData packet.

Multiple resource identifiers can be specified, causing each of the specified elements in the user registration database to be returned.

This packet is responded to with a series of packets (one for each request resource). These packets are either MDBP_ReadUserDataResp (if the resource was successfully read) or MCMP_Status (if there was an error reading the resource).

The MCMP Header of the read user data packet is:

PacketType:	MDBP_ReadUserData (h0007)
PacketSize:	0
ResourceID's:	1 or more
Shuntable:	No
ResourceReq:	Yes

READ USER DATA RESPONSE (MDBP_ReadUserDataResp): This packet is the response to an MDBP_ReadUserData packet. It contains the content of the requested element of the user registration database. The body is the data that was read. The MCMP Header is:

PacketType:	MDBP_ReadUserDataResp (h0008)
PacketSize:	Variable
ResourceID's:	None
Shuntable:	No
ResourceReq:	No

PURCHASING/ROYALTY REPORT (MDBP_PurchaseReport): One or more of

these packets is sent from the client to the server after the client has received a response to an MDBP_CreditReq packet, if the client has purchasing or royalty information to report.

The data area contains a generic royalty report header followed by a publication-specific royalty-report data area. In some cases, the royalty report header is sufficient to report the needed royalty information, so the data area is optional and does not have to be included. Any data following the royalty report header is assumed to be publication-specific data.

The MCMP Header of the royalty report packet is:

PacketType:	MDBP_PurchaseReport (h0009)
PacketSize:	Variable
ResourceID's:	None
Shuntable:	No
ResourceReq:	No

The Royalty Report Header is

```
RoyaltyReport,128
    . HeaderSize,2        $$ Size, in bytes, of royalty report header
    . Publisher,45        $$ Name of publisher
    . Publication,45      $$ Name of publication
    . VolIssue,20         $$ Volume and/or issue number of publication, if
                                  applicable
    . Version, 4          $$ Version of publication, if applicable
    . UserID,8            $$ ID of user who was reading this publication
    . CreditSpent,4,r     $$ Credit spent, in dollars, on this publication
```

DPP Packet Types

The distributed processing protocol (DPP) is used to reduce the processing load created by a particular task by distributing it over multiple computers. The protocol is used to search for and locate idle systems and (in conjunction with the CMXP protocol) transmit the appropriate code modules to those systems so that they can assist with the task. When the task is complete, the protocol is used to gather the results from all of the "assisting" machines and collect them and compile them on the machine that initiated the task.

The functionality provided by this protocol should not be confused with the load distribution functionality provided by the main MCMP protocol. The MCMP protocol's load distribution works by distributing client requests over various machines in various locations. The DPP protocol uses several machines working together to accomplish a single task, and is more suited to a local area network, and to processor intensive tasks, such as rendering of 3D images.

Each system involved in the distributed processing process must be configured with a list of those systems which can assist it with a task, as well as those systems which it can assist with tasks. This list can include entries with wildcards, to specify an entire network, such as 192.168.123.* for the entire 192.168.123. C-level network.

The purpose for this system configuration is to control who can utilize a system's processor. For example, a company might want to limit shared processing to systems within it's own internal network, for security reasons.

Systems can also be assigned priorities for access to a computer's processor. For example, a company may want all of it's computer to grant distributed processing requests from other computers on it's network in preference to other requests. However, if that company is affiliated with some other company, it might want to grant that other company access to it's computers for distributed processing purposes, provided that none of it's own

computers require processing assistance.

The following is a list of the packet types used by the DPP protocol at this time. The functionality of the DPP protocol can be expanded in future by adding to this list of packet types.

DPP_AssistReq	= 2	$$ Request for processing assistance
DPP_AssistResp	= 3	$$ Response to DPP_AssistReq
DPP_EndTaskReq	= 4	$$ Request to terminate processing assistance
DPP_EndTaskNotify	= 5	$$ Notification of termination of assistance
DPP_UpdateReq	= 6	$$ Request for update of task status
DPP_UpdateResp	= 7	$$ Response to UpdateReq

These packet types are described in detail below.

REQUEST FOR PROCESSING ASSISTANCE (DPP_AssistReq): This packet is sent by a system requiring processing assistance to another system to request processing assistance from that system. This packet contains all the information needed to initiate a distributed process, including the resource identifier for the initial code module to handle the process, so that the code module can be fetched via CMXP if necessary. The response to this packet is either a DPP_AssistResp packet (if the recipient system can assist) or an MCMP Status packet (if the recipient system can not assist).

Possible reasons for an MCMP_Status packet can include:

- Access Denied

  The system to which the packet was sent was not allowed to assist with the request. This is a result of the system generating the request not being listed in the appropriate configuration file on the receiving system.

- Insufficient Free System Resources

There are not enough free system resources on the system receiving the request for that system to assist with the distributed process. In some cases, a system may be to busy to even return this status value.

- Request Superceded

This indicates that the system had enough free processor time, but chose to assist a different system in preference to the one sending the request. The reason that Request Superceded is a separate status code from Access Denied is that "Access Denied" may generate an error if encountered by a program searching for systems to assist it (to notify the user of a possible mis-configuration) while Request Superceded would simply indicate that the system is not available to assist with the task at that given time, and would therefore not generate an error.

- Task-Specific Error

This is resumed by the code module that would handle the task. The MCMP_Status packet will contain an additional task-specific error code indicating the specific error which occurred. Task specific errors might include an error indicating that the system is not capable of assisting with the task due to a hardware limitation.

The packet body of the assistance request packet consists of a 32-byte header, followed by a task-specific data area, which contains any infommation that the code module referenced in the Resource ID requires to assist in the processing of the task. This could, for example, include an image (if an image must be processed) or a description of a 3D environment to be rendered.

The task-specific data area also contains information indicating which portion of the task the system is to work on (for example, starting and ending lines in the image) as well as the frequency with which the assisting system is to update the initiating system with processed data.

The MCMP Header of the assistance request packet is:

PacketType:	DPP_AssistReq (hO002)
PacketSize:	32 + Size of task-specific data
ResourceID's:	One (The ID of the code module to handle the distributed process)
Shuntable:	No
ResourceReq:	No

The DPP_ AssistReq Header is:

DPP_AssistReq Hdr,32		
	ProcessID,2	$$ Process Identifier
	,30	$$ Reserved

The elements of the DPP_AssistReq Header are described in detail below:

Process-ID: This is a 2-byte integer value that identifies the process. It is assigned by the system initiating the process, and is a unique identifier when combined with that system's IP address.

DPP_AssistResp: This packet is sent in response to a DPP_AssistReq to acknowledge that the system has begun assisting with the task. Because this is simply an acknowledgement message, there are no Resource ID's and there is no packet body. The MCMP Header is:

PacketType:	DPP_AssistResp (h0003)
PacketSize:	0

ResourceID's: None

Shuntable: No

ResourceReq: No

DPP_EndTaskReq: This packet is sent to an assisting system to instruct that system to cease assisting with a task prematurely (before the task is complete). This would be used, for example, if the user on the initiating system were to click a "cancel" button and abort the task. The MCMP Header is:

PacketType: DPP_EndTaskReq (h0004)

PacketSize: 16

ResourceID's: One (The ID of the code module to handle the distributed process)

Shuntable: No

ResourceReq: No

The **Packet Body** of the end task request header is:

DPP_EndTaskReq,16

. ProcessID,2 $$ ID of process to terminate

. ,14 $$ Reserved

DPP_EndTaskNotify: This packet is sent by an assisting system to notify the initiating system that it will no longer be assisting with a task. This is used both by itself, and as an acknowledgement to a DPP_EndTaskReq packet. This would be sent if, for example, the assisting system was to become too busy to continue to assist with the task, or if the assisting system was to be instructed by the initiating system to abort the task. This packet can also be used to notify an initiating system of a completed task. The MCMP Header is:

PacketType: DPP_EndTaskNotify (hO005)

PacketSize: 16

ResourceID's: One (The ID of the code module to handle the distributed process)

Shuntable: No

ResourceReq: No

The Packet Bodyof the end task notification packet is:

DPP_EndTaskResp,16

ProcessID,2 $$ ID of process to terminate

Tstatus,1 $$ Task status; 1=Complete, O=Incomplete (Aborted)

,13 $$ Reserved

DPP_UpDateReq: This packet is sent by the initiating system to instruct the assisting system to transmit processed data (a DDP_UpdateResp packet). For example, if an image was being processed, this would cause the assisting system to respond with the data making up the portion of the image that it has processed so far. The use of this packet type depends on the task. Some tasks will not use this packet at all, and will instead automatically generate DPP_UpdateResp packets at various intervals, and when the task is complete. The MCMP Header is:

PacketType: DPP_UpdateReq (h0006)

PacketSize: 1 6

ResourceID's: One (The ID of the code module to handle the distributed process)

Shuntable: No

ResourceReq: No

The Packet Body of the update request packet is:

DPP_EndTaskResp,16

ProcessID,2 $$ ID of process for which to return processed data

. ,14 $$ Reserved

DPP_UpDateResp: This packet is sent from an assisting system to an initiating system. It contains the data that has been processed so far as part of the task in question. For example, if an image is being processed, this packet would contain the portion of the image that had already been processed. Note that the data sent in these packets in not cumulative. That is, if two packets are sent in succession, the second contains only data not included in the f rst.

These packets are often sent in response to DPP_UpdateReq packets, although they can also be sent automatically by the program handling the task assistance, both during the task and upon task completion.

The packet body consists of a header, followed by task-specific data. All data not part of the header is assumed to be task-specific data.

The MCMP Header of the update response packet is:

PacketType: DPP_UpdateResp (h0007)

PacketSize: 16 + Size of task-specific data

ResourceID's: One (The ID of the code module to handle the distributed process)

Shuntable: No

ResourceReq: No

DPP_UpdateResp Header is:

DPP_UpdateResp,16

. HeaderSize,2 $$ Size, in bytes, of DPP_UpdateResp header

. ProcessID,2 $$ ID of process for which to return processed

data

,12 $$ Reserved

The term "code module" is used herein to denote a self-standing portion of an applications program dedicated to the performance of a specific operation of the applications program. For example, in a painting program, one code module may control the drawing of a line, while another code module implements the application of color and yet another code module is used for generating a geometrical figure such as a circle. These code modules are independent in that at least some of them are not required for executing any particular operation. Sometimes two or more modules are required to produce a particular result. However, in no case are all modules required.

The term "machine-executable" as used herein refers to code modules which are program modules, capable of controlling computer operations and changing the state of the arithmetic logic circuits of a general purpose digital computer, thereby changing the functionality of the general purpose digital computer. Thus, "machine-executable code modules" do not include data files and other passive electronically encoded information.

The term "applications program" as used herein refers to any executable collection of computer code other than operating systems and other underlying programming for controlling basic machine functions. Thus, the Modularized Code Master Protocol, including its subprotocols, is an applications program which is itself modularized and transmittable in code modules over a network. For example, at least some subprotocols will not exist on some secondary servers. Should such a subprotocol be required for a secondary server to compete a task or process a user's request, then that required subprotocol may be transmitted over the network from the primary server to the secondary server.

Subprotocols are handled by including a subprotocol identifier in the MCMP Header

attached to each MCMP packet. On an MCMP server, the circuits for handling the subprotocols may be plug-in modules which are handled in real-time by the CMXP protocol. When an incoming packet is received, the appropriate subprotocol handler is called. The subprotocol handler can then process the packet in whatever way is required. A protocol handler becomes involved in the load distribution process because the MCMP server has no way of knowing what format the resources are in, how to transfer them between servers, or what the caching rules are. The subprotocol handler must deal with accessing, transfer, and caching of the protocol-specific resources. The subprotocol handlers are called periodically during the MCMP server's main processing loop, allowing it to perform various maintenance tasks. Alternatively, the subprotocol handler could be called from a loop running in a separate thread.

The handler for a specific subprotocol may request that the MCMP server flag a socket as being in a Proprietary Dialog Mode (PDM). On a PDM socket, all incoming data is passed directly to the subprotocol handler without being processed by the MCMP server. When a socket is retunred to normal operation from the PDM operation, the subprotocol handler must pass any "extra" unprocessed data to the MCMP server, since it may have read a portion of one or more MCMP packets.

The term "primary server" or "source server" is used herein to denote the authoritative source for the resources relating to a particular application. For example, the primary server for the TenCORE Net Demo would be the server where the latest version of the demo was always posted.

The term "secondary server" as used herein denotes a server that receives service-handoffs from a primary server. A secondary server usually mirrors the content of the primary server for which it is secondary. For example, a secondary server for the TenCORE

Net Demo would be a server that could take over servicing clients if the primary server became too busy. A secondary server does not necessarily contain mirrors of the resources for which it is secondary server, inasmuch as the secondary server can request these resources from the primary server as needed.

A single machine is can be both a primary server and a secondary server. For example, a machine could be primary server for the TenCORE Net Demo, and secondary server for the Country Closet Clothing Catalog.

A single machine can function as primary server for multiple applications, and can function as secondary server for multiple applications.

The word "resource" is used herein to denote any block of data that can be used by a server or client. A resource can be a file, a code module, a portion of a file, a code module, a portion of a code module, a file directory, a code module directory, or any related piece of information, including the CMXP Prohibited List. The term "resource" is also used to refer to hardware and system resources, such as processor time and memory.

The term "tree-file" refers to a Catharon Tree-Structure Nameset File. A tree-file contains a series of named sets of records, which are grouped and nested into a tree-like structure (similar to an operating system's file system). In tree files, names beginning with a percent sign "%" are reserved for internal use. Any other names may be used by whatever application is maintaining the tree-file. Currently, only one percent-sign name has been assigned. It is "\%System" and it contains general information about the file, including (optionally) the name of the application that created the file, the user under who's network account the file was last edited, the date and time the file was last edited, the location of various resources in the file, the location of the default folder (if none specified), and the file's associated information.

As related hereinabove, TenCORE is an interpreted language which utilizes pseudocode. Interpreter programs suitable for use with the TenCORE programming language as modified in accordance with the above descriptions are in common use today.

A description of the TenCORE programming language is set forth below. The basic characteristics of the language are discussed first, then the treatment of variables. Finally, an exposition is made of all the important commands used in the language. From this information, as well as the foregoing description, one of ordinary skill in the art can generate a modular programming language suitable for use in the invention.

TenCORE Language Basics

The TenCORE Language Authoring System is a complete programming environment specially enhanced for implementing computer-based training. Its editors aid in the creation of source code, images, fonts, screen displays and data manipulation. The language has complete facilities for display creation, response input and analysis, and data manipulation within a structured programming environment.

Command Syntax

The primary building block of the language is the **command**. TenCORE contains about 175 commands mnemonically named for the functions they perform. Most commands are followed by a **tag** often with keywords that furher define the specific function desired. A command and tag taken together is called a **statement**. As in any language, there are rules that determine the syntax of a tag and how a sequence of statements interact to perform a specific task.

The TenCORE language is a fixed field language. In the simplest form, the syntax has the form:

command tag

The *command* field contains the name of a command of up to 8 characters long. The *tag* field begins at character position 9 (there is a tab stop here in the source editor) and can be up to 119 characters in length on the remainder of the line. For many commands, the tag can be continued for several lines by tabbing over (leaving blank) the command field on the following lines. Some typical lines of code look like this:

```
at        5:10
color     yellow
write     Welcome to Basic Sign Language
```

```
Today's lesson consists of...
```

Each statement begins with a command and is followed by a tag. In the first line, the **at** command causes the cursor to be positioned on the fifth line and tenth character position of the display. The second **color** command selects yellow to be used for following graphics and text. The last **write command** puts text on the display at the position and color previously specified. The **write** statement's tag continues over several lines by tabbing over succeeding command fields.

Selective Form

Many commands have a selective form:

command SELECTOR, negTAG, zeroTAG, oneTAG, . . .nTAG

The *SELECTOR* is an expression consisting of variables, constants or calculations that is evaluated and used to select a specific tag from the list. The tags are usually separated by semicolons although other separators are available. The *nTAG* case is selected when *SELECTOR* evaluates to *n* or greater. A blank entry in the list (..) can be used to skip execution of the command for a specific case. If *day* is defined as an integer variable programmed to hold a particular day of the week, then the following statements would display that day on the screen

```
at        5:10
write     Today is
writec    day;;; Monday; Tuesday; Wednesday; Thursday;
          Friday; Saturday; Sunday
```

Since TenCORE defines logical *true* to be the value -1 (or any negative value) and *false* to be 0 (or any positive value), any two tag selective acts as a true or false decision:

command SELECTOR; trueTAG; falseTAG

BNSDOCID: <WO___ 9907007A2_I_>

Conditional Form

Some commands (especially the judging commands) have only a *CONDITION* as the tag that is used to determine if the command executes or not. The condition is any expression that logically evaluates to *true* (-1) or *false* (0).

command CONDITION

For example, if you wanted to accept misspellings after a student's third attempt at matching a response, you could use a conditional expression based on the system variable *ztries* which holds the number of previous judging attempts.

```
okspell ztries > 3
answer Mississippi
```

Embedding

Many text handling commands allow the embedding of further commands in the body of the text

command TEXT «command.tag» TEXT

The symbols « and » are the embedding symbols accessed by the [ALT][.] and [ALT][ ] keys Frequently, the embedded command name can be abbreviated to the first letter or two to save space: e g., the abbreviated form of the embedded **show** command is simply **s**. Embedding is frequently used for display of variables and the control of plotting attributes within text statements thereby effeciently coding for an entire display:

```
write    «color,red»Your score «color,white» is «show,score»%.
         The «c,green»class average «c,white» is «s,average»%.
```

Comments. Spacing and Continued Commands

An asterisk (*) at the beginning of a line marks the entire line as a comment that is removed by the compiler and has no effect on execution. A comment can be placed on any line of

BNSDOCID <WO 9907007A2 I >

code by placing it after double dollar signs $$: the compiler removes the comment and it does not affect the execution of the code:

```
calc    temp <= temp*9/5 + 32      $$ convert centigrade to Fahrenheit
```

Normally, trailing spaces on a tag are removed by the compiler and do not affect execution. This is also true when double dollar $$ comment signs are used: the compiler removes the comment and any spaces back to the real tag. For some text commands, you may actually want the trailing spaces and can direct the compiler to maintain them by using an on-line comment with triple dollar signs $$$. The following example would display something like *Welcome Bob:*

```
write   Welcome $$$    maintain single trailing space
showa   name
```

Most commands with non-selective tags can be repeated again without the need for typing in the command field again. "Space" can also be added between most lines of code to make the coding read well:

```
*       This section of code initializes parameters.
calc    frame <= 2045          $$ starting frame number
        file <= 'bangs'        $$ use the explosion library
        block <= 'mega'        $$ use the big explosion
        type <= 'nameset'      $$ these are nameset files
attach  file;type              $$ attach the file
```

Units

A sequence of TenCORE statements makes up a functional entity called a *unit* analogous to a procedure or subroutine in other languages.

Commands can be grouped into units in any way that makes sense. Simple "page turning" lessons consist of multiple frames of material: each could be a unit. A complex graphic that is

BNSDOCID: <WO_ _9907007A2_I_>

used several places in a lesson can be put once into a single unit which is then referenced at each point it is needed. A simulation can have each functional part in its own unit.

Each unit must have a unique name of up to 8 characters in length starting with a letter. Punctuation, special symbols and the two system reserved names **x** and **q** are not allowed.

Referencing Units

If you want to reference a unit in the same lesson, you simply state the unit's name:

```
do        clock          $$ call unit clock to start timing going
```

If you want to access a unit in a lesson different from the one currently executing, then you must give both the lesson and unit names.

```
do        maps,illinois  $$ show the map of Illinois
```

Finally, you may not want to explicitly state the unit or lesson names but rather refer to names that have been calculated into variables This is done by *embedding* the variables where the explicit names would normally appear. Embedding consists of surrounding the variables with the « and » symbols. For example, say that all the lesson and unit names in a project have been put into the 8-byte arrays *lessons* and *units*, then any unit can be accessed by setting the indices to point to the desired unit:

```
do      «lessons(i)»,«units(j)»
```

Generic Unit Names

System keyword names exist to provide generic branch destinations to main units in the current lesson or back to the system. For example,=**next** and =**back** can be used for the unit name when **jumping** to the next or previous main unit in the current lesson; =**exit** can be used to exit the current lesson back to DOS, the editor or the system Activity Manager. A descriptive list of the nine generic unit names along with examples of their use is found with the **jump** write-up.

ENSDOCID <WO 9907007A2 I >

Unit Terminology

Units can play a number of different roles during execution depending on how they are used. Units may correspond to major displays and interactions that a user experiences: the "pages" of the lesson. On the other hand, a unit may simply be a subroutine called by one of these major units to perform a sub-task. The following terms are used to describe various functional units.

main unit

A main unit is the first unit executed in a lesson and any other unit reached by a **jump** type branch. It normally corresponds to one "page" or user situation in the lesson. It is the starting point for all user interaction. When the end of the main unit is reached in execution, the system waits for a key press or other interaction to occur to branch the user to another main unit Branching to a new main unit normally:

- erases the screen
- re-initializes all plotting parameters
- resets branches to their lesson defaults
- clears all pointer areas
- establishes a new main reference point in the lesson

current unit

The **current** unit is the unit currently executing. It may be the main unit or a subroutine unit called from the main unit by a **do** or **flow** command.

done unit

A **done** or called unit is used to describe a unit executed as a subroutine. When the end of a done unit is reached, control returns to the command following the calling command in the invoking unit or to the waiting state from where a **flow** branch was triggered.

BNSDOCID: <WO 9907007A2 I >

base unit

The **base** unit is the main unit at the time a branch occurred that was modified with the **base** keyword. It can be used to set up a common help sequence that can automatically return to whatever main unit called it. Return to the base unit occurs through a branch using the **=base** generic name.

startup unit

The **startup** unit is the first unit executed in your lesson either from DOS or a router such as the system Activity Manager. It is usually the first physical unit in the lesson file and should contain a **startup** comrnand if the unit is to be directly entered from DOS.

restart unit

A **restart** unit is placed in the lesson flow where return from an interruption in the learning session can occur. It would generally have a **restart** command in it and should have a display that can re-orient the student after the interruption. The system Activity Manager would branch a user to one of these units if the "continue last lesson" option is chosen upon student signon.

Control Blocks

Control blocks offer a means to expand control of major events such as starting or quitting a lesson or in going from one main unit to another. Coding in a lesson's control blocks is automatically executed at these major events regardless of the specific starting, ending or main unit involved. This is a convenient way to ensure that necessary initializations or cleanups are done. Control blocks are created in the editor on the block directory page. Besides the following, some additional control blocks are discussed in the **Updates** chapter.

BNSDOCID <WO 9907007A2 I >

+initial

The code in a **+initial** control block is executed each time an outside **jump** type branch is made to the lesson: e.g., starting the lesson from **DOS**, executing the lesson from the editor, jumping to the lesson from another lesson or from the Activity Manager. It can be used to load fonts, set plotting conditions, initialize data keeping, etc. regardless of where entry to a lesson occurs.

+main

A **+main** control block is executed each time a new main unit is entered: it is used to perform operations common to the beginning of all the main units in a lesson. For example, it can be used for: displaying a lesson-wide background image and flow bar, saving restart information, or displaying debugging information during lesson editing such as the current main unit name.

+exit

A **+exit** control block is executed whenever control leaves the current lesson as during a **jump** to a different lesson or when quitting to DOS or to the calling program that started the current lesson running. It can be used, for example, to collect end-of-lesson summary information or to turn off any devices that were turned on for the lesson.

+editor

A **+editor** control block is executed each time you go to edit the file. It can be used, for example, to load fonts for text editing.

Screen Resolution

A wide range of **PC** display hardware is supported from the original **CGA** and **EGA** adapters to **MCGA**, **VGA** and Enhanced **VGA** adapters. Each of these includes an increasing collection of graphic and text screen resolutions and color capabilities. TenCORE courseware is typically created using a specific screen resolution and color range that is supported on the target

BNSDOCID: <WO__9907007A2 I >

population of run-time machines for your courseware: often the lowest common denominator is used. The **screen** command selects these parameters and is usually placed in the **+initial** control block of a lesson.

Graphic Coordinates

A particular screen pixel is addressed by x and y graphic coordinates with 0,0 specifying the lower left corner of the screen:

```
dot      100,    200              $$ write the pixel at x=100, y=200
at       320,    240              $$ at the center of the vga screen
circle,  75,fill                  $$ draw a filled circle of radius 75
```

Most display commands update the system variables **zx** and **zy** as part of their operation and thereby define the current screen location.

Character Coordinates

Text can be more conveniently addressed by character coordinates that specify a line number and character position that are separated by a colon to distinguish them from graphic coordinates:

```
at          5:15        $$ line 5 character 15
write    Text on line 5....
```

Defaults

Graphics and text commands display using the current settings for numerous attributes such as: the foreground and background colors, the plotting mode, text size and drop shadowing, etc. These attributes can be set just prior to using them:

```
color    yellow          $$ the following in yellow
mode     write           $$ overstrike plotting mode
text     shadow; on      $$ drop shadowing on text
```


ENSDOCID <WO 9907007A2 I >

```
text      size; 2          $$ size two fonts
at        5:10             $$ at line 5, character 1
write     A cross section of the sun...
```

Attributes have initial default values that are set by the system when executing an **initial** or **screen** command as at the start of a lesson: e.g., the system default foreground color is white and the text size is 1. Attributes are automatically returned to their default values at the start of each main unit or they can be forced as a group to their default values at any time by use of the statement **status restore: default.** Upon **jumping** to a new main unit after executing the above example code, size 1 text and color white would be put back in operation. See the **initial** command for a list of the display attributes and their standard default values.

To give a common "feel" to a lesson, it is convenient to set the attribute values to your default values at the start of all units in the lesson. That way, if you later change your mind about using, say, the gothic font throughout a lesson, you can merely change the font default value to something else and have it apply to the entire lesson The **status save: default** statement is used to reset the attribute defaults to their current values:

```
color     yellow
text      spacing; variable
text      shadow, on
text      margin; wordwrap
status    save; default          $$ the above are now lesson defaults
```

Lesson defaults are usually set in the **+initial** control block of a lesson.

Variables

Author-Defined Variables

TenCORE supports both local and global variables.

BNSDOCID: <WO 9907007A2 I >

Local variables are defined in a specific unit and are available only to that unit. When a unit is exited, either by reaching the end of the unit or by branching to a new main unit, the values of its local variables are lost.

Global variables are available throughout the lesson in which they are defined and retain their values for the entire program. With the help of the TenCORE Activity Manager or a similar program, global variables can even be made to retain their values across TenCORE sessions.

Local Variables

Local variables are created by defining them within a source unit, between a **define local** and a **define end** statement.

```
define   local
height,2    $$ defines the 2-byte integer variable height
length,4,r  $$ defines the 4-byte real variable length
define   end
```

Variables defined in this way are known only to the unit in which the definition occurs, and the values assigned to these variables are lost when the unit is exited via any command which goes to a new main unit.

Local variables are usually used for temporary information which is not needed outside the current unit, such as loop indexes.

Global Variables

A special block, called the **defines** block, is normally used to hold global variable definitions. Within the **defines** block, variables are defined by inserting lines of this form:

name,size[,type]

as in:

ENSDOCID: <WO 9907007A2 I >

defines

```
height,2 $$ defines the 2-byte integer variable height
length,4,r  $$ defines the 4-byte real variable length
```

Variables defined within a **defines** block are global variables. They are available in every unit within the file, and are normally used to store informafion which is needed in several places, such as the user's name or status within the program.

Global variables for every lesson are stored from the beginning of a single memory area reserved for global variable storage. This means that if a program extends over two or more source files. it is important that all source files use the same defines and/or coordinate their shared use of this area. This subject is treated in more depth in the Physical Allocation of Variables chapter

Local Definition of Global Variables

Similar to local variables. global variables can be defined between a **define global** and a **define end** statement.

```
define   global
height,2
length,4,r
define   end
```

Variables defined between **define global** and **define end** use the global variable storage area. but are known only within the unit in which the definifion occurs.

The **define global** command should be used only for isolated test units and a few special purpose utilities. Used without an understanding of how space for global variables is allocated. the **define global** command can lead to conflicting definitions of variables and unpredictable program behavior. For more information, see the Physical Allocation of Variables chapter.

BNSDOCID: <WO 9907007A2 I >

Precedence of Variable Names

A local variable can be given the same name as an existing global variable· in this case, the local variable supercedes the global variable.

globals (defines block)

```
user,20
screen,5

define   local
user,4,r
define   end
```

Here "user" has been defined as both a global and a local variable. Within unit **screen5**, the global variable "user" cannot be accessed Any time that "user" appears in unit **screen5**, it refers to the local variable

Variable Names

Variable names can be one to eight characters long. Any of the following may be used in variable names:

- letters (including those in the extended character set, such as Ä and Ö)
- numerals
- period (.), underline (_), caret(^) and tilde (~)
- extended ASCII characters

The first character of a variable name may not be a numeral or a period. Some samples of valid variables names are:

```
username
Lesson
xLOCAT
```


ENSDOCID <WO 9907007A2 I >

```
x.loc
last_x
list
root
```

Invalid variable names include:

```
2student     may not start with a numeral
useraddress        name longer than 8 characters
unit'n       punctuation not allowed
```

Case is significant: the variable "a" is different from the variable "A" and the variable "alpha" is different from the variables "Alpha" and "ALPHA".

Spacing around the elements of a variable definition is not significant. Most authors apply one of two different styles in defining variables

```
define   local
width,2
height,2
area,2
length,2
mass,4,r
define   end
```

In the style above, the length and type of each variable are written immediately after the variable name. Another common style is the following:

```
define   local
width    ,2
height   ,2
area     ,2
length   ,2
mass     ,4,r
```


BNSDOCID: <WO 9907007A2 I >

```
define   end
```

In this style, the lengths of all the definitions are aligned in a tabular format

These styles are functionally equivalent.

Variable Types

Any of the three types can be defined globally or locally. Variables in TenCORE are not "strongly typed", i.e. a variable defined as one type can often be used in operations primarily intended for another type of variable.

Integer Variables

Integer variables store positive and negative whole numbers. Integer variables are defined as follows:

```
a,1
x,2
line,3
temp,4
total,8
```

Where the normal form is *name.size*. The size of an integer variable can be 1, 2, 3, 4, or 8 bytes. The size determines the range of values that the variable can store.

size	Range
1	-128 to 127
2	-32768 to 32767
3	-8,388,608 to 8,388,607
4	-2,147,483,648 to 2,147,483,647
8	$-9*10^{18}$ to $-9*10^{18}$

Integer variables are significant for all values within their range: no rounding errors occur from addition, subtraction, or integer multiplication of integer values. However, rounding is performed when real values are assigned to integer variables and when integer division is performed.

SNSDOCID <WO__9907007A2 I_>

Integer-type variables optionally take the *type* modifier **integer**, which can be abbreviated as **i**:

```
x,2,i
line,3,integer
```

TenCORE stores integer values in natural (high-byte to low-byte) order. (Many MS-DOS programs store integer values in low-byte to high-byte order.)

Real Variables

For storing non-integer numeric values, TenCORE provides *real* variables, which are defined as follows:

```
force,8,real
area,4,r
```

Where the normal form is *name.size.***real**. Real variables can be either 4 or 8 bytes long, and the tag **real** or its abbreviation **r** must follow the size specification when the variable is defined.

The size of a real variable determines the range of values that the variable can store and the *precision*, measured in *significant digits*, with which the variable can store the value.

Size	Range	Significant Digits
4	$10^{\pm 38}$	6
8	$10^{\pm 308}$	15

Eight-byte real variables should be used whenever storage space is not at a premium. If 4-byte real variables must be used, they should be compared only to other 4-byte real variables. Comparisons of the form **if length = 1.1** can give unexpected results due to the limited precision of 4-byte real variables.

Real variables are stored in IEEE format. As in the case of integer variables, the bytes are stored in natural (high-to-low) order rather than the reverse order used by some programs.

SNSDOCID: <WO 9907007A2 I >

A variable declared with no size and no type is created as a 4-byte real variable.

Non-Numeric Buffer Variables

For holding text or other information, non-numeric variables can be defined with any size from 1 byte to 32767 bytes (provided that enough variable space is available), as in:

```
username,20
unitname,8
italics, 2052
```

The general format is *name,size*. Non-numeric variables are most often used to store text, but they can also be used to store any other information requiring a buffer.

There is no difference between the definition of a non-numeric variable with a length of 1, 2, 3, 4 or 8 bytes and the definition of an integer variable of the same size. Only the context in which the variable is used indicates whether the variable holds integer or non-numeric data.

In TenCORE command syntax, non-numeric variables are referred to as "buffers".

Arrays

The variable types described so far are all *scalar*, meaning that each variable holds only one value. TenCORE also supports *array* variables. An array is a series of variables of the same size and type, all referred to by the same name, using an *index* to specify individual elements of the array. An array index is simply a number in parentheses which appears immediately after the name of the array.

```
average(100),4,real     $$ array of one-hundred 4-byte reals
count(5),2,i            $$ array of five 2-byte integers
bignum(20),8            $$ array of twenty 8-byte integers
table(10),256           $$ array of ten 256-byte non-numeric variables
```


ENSDOCID: <WO 9907007A2 I >

The definition of an array variable looks like the definition of a scalar variable, except that the variable name is followed by a number in parentheses which specifies the number of elements (values) the array should hold. The general form for defining an array is:

name(elements),size, type

Arrays can then be referenced in TenCORE code as in:

```
calc      count(3) = 10
datain        1,table(index),1
```

Brackets are synonymous with parentheses, permitting the following alternative notation.

```
calc      count[3] = 10
datain    1,table[index],1
```

TenCORE supports only one-dimensional arrays, arrays of two or more dimensions can be simulated using functions

System-Defined Variables

TenCORE provides a number of system-defined variables which store information about the current state of the program. These system-defined variables (usually shortened to system variables) are simply names which TenCORE recognizes as representing numbers or other variable information.

System variables provide information which is frequently needed or which might be tedious or impossible for the author to maintain in author-defined variables

For instance, the system variable **ztries** can be used to refer to the number of times the user has entered a response to the currently active arrow structure:

```
if        ztries   > 3
.         write    The correct answer is: green
endif
```


BNSDOCID: <WO__9907007A2_I_>

This example displays the correct response to a question if the user has already tried to answer more than three times.

System variables all begin with the letter "z". This makes them easier to recognize. As a matter of good programming style, author-defined variables should therefore avoid names which begin with "z". It is possible to create an author-defined variable with the same name as a system variable. In this case, the author-defined variable has precedence and the value of the system-defined variable becomes unavailable

The following are examples of some other system variables

- **zreturn** is set by many commands to indicate the success or failure of the command. A negative value of zreturn indicates success, while various non-negative values indicate different reasons for failure of the command **zreturn** is a one-byte integer reflecting *command status.*
- **zcolor** is set to the value of the current foreground plotting color. It is a two-byte integer relating to *display control*
- **zmainu** contains the name of the current main unit It is an eight-byte variable relating to branching

Values cannot be directly assigned to system variables. However, some system variables are affected by an associated command; for instance, **zcolor** indicates the currently selected foreground color and is therefore affected by the **color** command. Executing **color white+** sets **zcolor** to 15, the value corresponding to bright white.

A comprehensive list of system variables organized according to category follows. In addition, many command descriptions refer the reader to system variables affected by the command.

BNSDOCID: <WO 9907007A2 I >

System Variables

zareacnt 2(int) Number of areas currently defined

zareahl 2(int) Area ID of currently highlighted area (0 if no area highlighted)

zargs 1(int) Number of non-null arguments actually received by either a -receive-or by the return argument list of a -do

zargsin 1(int) Number of arguments sent by -do-, jump-, or -jumpop

zargsout 1(int) Number of arguments the invoking unit expects returned

zaspectx 2(int) x-aspect ratio correction factor

zaspecty 2(int) y-aspect ratio correction factor

zauthsys 1(int) Indicates the TenCORE executor type:

-1 = LAS author executor (TCAUTHOR.EXE)

0 = student executor (TCRUN.EXE)

1 = Producer executor (TPR.EXE)

zbinary 1(int)Type of file from which "zlesson" was loaded

-1 = binary, 0 = LAS source, 1 = Producer source

zcharb 2(int) Offset of standard size 1 font baseline

zcharh 2(int) Height in dots of standard font or charset

zcharw 2(int)Width in dots of standard font or charset Note. all text positioning is based on the standard font (zcharw, zcharh, zcharb), not the current font.

zclipx1 2(int) X-coord of lower left corner of clipping area

zclipx2 2(int) X-coord of upper right corner of clipping area

zclipy1 2(int) Y-coord of lower left corner of clipping area

zclipy2 2(int) Y-coord of upper right corner of clipping area

zclock 4(int) Value of the system clock, accurate to ~55ms

BNSDOCID: <WO___9907007A2_I_>

zcolor 2(int) Foreground color set with -color

zcolore 2(int) Erasure color set with -colore

zcolorg 2(int) Background color set with -colorg

zddisk 1(int) DOS default drive: 1=A:, 2=B:, 3=C:

zdisk 1(int) Method of searching for data files: Values same as "zedisk"

zdisks 4(bits)Bitmap of drives TenCORE can access, set in DOS by SET CDISKS=

zdisplay 1(int) Current active display number (dual screen driver only)

zdispx 2(int)X-size of current window

zdispy 2(int)Y-size of current window

zdolevel 2(int) Current level of do type command stack

(do, library, flow do, flow library)

zdomcnt 2(int) Number of existing domains

zdomname 8(alpha)Name of current domain

zdompar 8(alpha)Name of parent domain

zdoserr 1(int) DOS disk error code Set only when "zreturn" = 0, indicating a disk error

0 disk is write-protected

2 rive not ready (door open or bad drive)

4 CRC error (error detected in data)

6 seek error (bad drive or diskette)

7 unknown media type (unrecognized diskette)

8 sector not found (bad drive or diskette)

12 general failure (bad drive or diskette)

zdosver 4(alpha) DOS version number in the format "X.XX"

zedisk 1(int) Method of searching for source/binary files:

3NSDOCID: <WO_ _9907007A2_I_>

-1 = search all drives; 0 = search only "zddisk"

-2 = search only system pseudo-drive

1, 2, 3,... = search only drive A:, B:, C:

zedoserr 8(int) DOS extended error code, set when "zreturn" = 0. Valid only when running under DOS 3.0 or higher. The format is:

2 bytes error code

1 byte error class

1 byte actions

1 byte locus

The remaining three bytes are reserved.

zenable 4(bits) -enable- flags

This variable is a set of bits numbered 1 to 32. Certain its correspond to -enable- commands as follows

1	enable absolute	9	(reserved)
2	enable pointer	10	(reserved)
3	enable mode	11	(reserved)
4	enable cursor	12	enable break
5	enable ptrup	13	enable arrow
6	enable font	14	(reserved)
7	enable area	15	enable fillbreak
8	(reserved)	(bits 14-32 reserved)	

Use the bit() function to test "zenable". For instance, bit(zenable,2) is -1 if the pointer is enabled and 0 if not

zenvlen 4(int) Length in bytes of DOS environment buffer

BNSDOCID: <WO___9907007A2_I_>

zenvloc 4(int) Absolute memory location of DOS environment buffer

zerrcode 2(int) "error code" value from "zedoserr"

zerrlevl 2(int) DOS variable "errorlevel" as return from last exec file.- command

zerrorl 8(alpha)Lesson name of unit to execute if an execution error occurs, set by -error

zerroru 8(alpha)Unit name of unit to execute on execution error

zexitbin 1(int) Type of file from which =exit branch will be loaded:

-1 = binary; 0 = source; 1 = tpr

zexitl 8(alpha) Lesson name to branch to on -jump =exit

zexitu 8(alpha)Unit name to branch to on -jump =exit

zfcrecs 4(int) Number of contiguous free records in the attached nameset type file

zfdisk 1(int) Disk on which the attached file resides.

1 = drive A:, 2 = drive B:, etc.

-2 = system path

zflowcnt 2(int) Number of flow branches currently defined

zfname 8(alpha)Name of the attached file. 0 if none attached

zfnames 2(int) Total number of names in the attached file; has no meaning for dataset files

zfontb 2(int) Offset of current font baseline

zfontf 4 (bits)For font groups, specifies which of the attributes were fit correctly and which were synthesized. Bits defined as follows:

6 = Bold attribute matches request

7 = Italic attribute matches request

8 = Size attribute matches request

16 = Narrow attribute matches request

22 = Bold Synthesized

BNSDOCID: <WO__ 9907007A2_I_>

23 = Italic Synthesized

24 = Size Synthesized

For the bits above that are synthesizable, the attribute may not match the request but also may not be synthesized; for example, if the font group only has bolded items, and the current attributes specify non-bolded, then bits 6 and 22 will both be off

zfonth 2 (int)Height in dots of currently active font

zfontret 1 (int)Information on how closely the selected font fits the requested parameters. All values available for font groups, For fonts. values -1. 2 and 3 are possible. Set by -font-, -text-,status restore-.

-2 = partial match. some synthesis required

-1 = exact match

= partial match. some non-synthesizable attributes could not be matched

1 = no suitable match found; base font is in effect

2 = base font not available; standard font is in effect

3 = standard font not available; charsets are in effect

zfontw 2(int) Width in dots of currently active font

zforce 2(bits)force- flags

This variable is a set of bits numbered 1 to 16. Certain bits correspond to -force- commands as follows:

1 force cursor lock-zinfo 8(int) Associated information bytes for the selected name in the attached file; has no meaning for dataset files

2 force number lock-

3 force extended-

BNSDOCID: <WO 9907007A2 1 >

4 force charset-

(bits 5-16 unused)

Use the bit() function to test "zforce". For instance, bit(zforce,1) is -1 if -force cursor lock- is in effect and 0 if not

zfrecs 4(int) Total number of data records in the attached file

zfrombin 1(int) Type of file from which "zfroml" was loaded:

-1 = binary; 0 = source; 1 = tpr

zfromd 1(int) Drive from which "zfroml" was loaded. Same values as zldisk

zfrom 8(alpha)Lesson name of unit which invoked current unit via -do-, -library-, or main-unit branch

zfromlin 2(int) Line number of command in "zfromu" which which invoked the current unit

zfromu 8(alpha)Unit name of unit which invoked current unit

zfrecs 4(int) Total number of data records in the attached file

zftype 8(alpha)Type of the attached file. 0 if none attached; file types: source, tpr, dataset, nameset,binary, roster, course, studata, and group

zfunames 2(int) Number of names used in the attached file

zfurecs 4(int) Number of records used in the attached file; has no meaning for dataset files

zfver 4(alpha)TenCORE version which created the attached nameset as "X.XX"

zhcolor 2(int) Hardware color set set with -color

zhcolore 2(int) Hardware erasure color set with -colore

zhcolorg 2(int) Hardware background color set with -colorg

zhdispx 2(int) Physical x-size of current screen type

ENSDOCID <WO__9907007A2_I >

zinfo 8(int) Associated information bytes for the selected name in the attached file; has no meaning for dataset files

zinput 2(int) Keypress value of last key processed

0 no input

1- 255 standard or extended character

> 255 non-character keypress

"zinput" contains 2-byte keypress values. To determine if "zinput" contains a particular value, one of the following forms is required:

zinput=%"character" zinput=%alt"character"

zinput=%ctl"character" zinput=%altctl"character"

zinput=%keyname

The %alt, %ctl, and %altctl modifiers operate on single standard ASCII characters (h20 - h7f). The % modifier operates on standard and extended characters (h20 - h0ff) in double quotes and on named keys

zinputa 2(int) Area ID of area causing the current zinput value (0 if zinput was not generated by an area); set when zinput updated

zinputainfo 8(int) Value of info tag for area that caused this branch

zinputf 4(bits Keyboard and pointer status at last input

This variable is a set of bits numbered 1-32. Certain bits correspond to keyboard and pointer status as follows:

1 Insert turned on

2 CapsLock turned on

3 NumLock turned on

4 ScrollLock turned on

BNSDOCID: <WO 9907007A2 I >

5 either Alt key down

6 either Ctrl key down

7 left Shift key down

8 right Shift down

9 SysReq key down

10 CapsLock key down

11 NumLock key down

12 ScrollLock key down

13 right Alt key down

14 right Ctrl key down

15 left Alt key down

16 left Ctrl key down

17 enhanced keyboard "extra" key

18 key is from Alt+numbers

19 key is from pointer action

20 key is from "pointer up" action

21 key is from touch device

(bits 22-29 reserved)

30 middle mouse button pressed (Some mice produce bits 31 & 32 instead of 30 on a middle button press)

31 right mouse button pressed

32 left mouse button pressed

Use the bit() function to test "zinputf". For instance, bit(zinputf,2) is 1 if CapsLock was turned on during the input in "zinput", and 0 if not.

ENSDOCID <WO 9907007A2 I >

zinputx 2(int) Graphics x location of pointer when "zinput" was last updated

zinputy 2(int) Graphics y location of pointer when "zinput" was last updated

zintincr 2(int) Minimum significant increment for I value in -palette

zintnum 1(int) Interrupt number used by TenCORE device drivers

zkeyset 2(int) shift flag byte and scan code of last key processed.

high byte = shift flags

low byte = scan code

The high byte is a set of bits numbered 1-8. duplicating the first eight bits in "zinputf"

The low byte is a "scan code" identifying a physical key

zldisk 1(int) Drive from which the currently executing lesson was loaded.1,2,3...=A:,B:,C:...; -2=system drive

zlength 2(int Length of judging buffer after -put

zlesson 8(alpha) File name of current unit

zlident 8(alpha) Identification stored in current binary file

zline 2(int) Current character line

zlinfo 2(int) Number of bytes of associated information for the attached file; has no meaning for dataset files

zlname 2(int) Maximum length of a name in characters for the attached file; has no meaning for dataset files

zmainl 8(alpha) File name of main unit

zmainu 8(alpha) Unit name of main unit

zmargin 2(int) x-coordinate of left text margin

zmaxarea 2(int) Maximum number of areas; default=1000

BNSDOCID: <WO__9907007A2_I_>

zmaxcol 2(int) Maximum number of colors available on current display

zmaxdom 2(int) Maximum number of domains allowed

zmaxflow 2(int) Maximum number of flow branches that can be defined

zmaxpage 2(int) Number of available hardware display pages

zmlength 4(int) Size, in bytes, of the last-referenced memory block.

Due to Windows memory management and disk swapfile, the effective memory pool size is generally more than large enough for most TenCORE needs on a typical Windows system. However, you should always check zreturn because it is possible for memory-related commands to fail, particularly on Windows systems with low memory. For compatibility reasons, the system variables zmem, zrmem and zfmem never return a value greater than 512K (524,288). However, if enough memory is available, a memory block as large as 1,048,560 bytes (hex FFFF0) can be created. Also for compatibility reasons, zxmem always returns a value of 0.

zmem 4(int) Current size, in bytes, of memory pool

zfmem 4(int) Largest block that could be allocated in the memory pool

zrmem 4(int) Largest block that could be allocated in the memory pool

zxmem 4(int) Set to 0

zmode 2(int) Current screen display mode

0 = inverse 3 = write 6 = add

1 = rewrite 4 = noplot 7 = sub

2 = erase 5 = xor

zmouse 1(int)Indicates if the mouse device driver is loaded:

-1 = mouse driver loaded;

0 = not loaded

BNSDOCID. <WO__9907007A2_I_>

zmstart 4(int) Absolute address of the last-referenced memory block. The top two bytes are the segment and the bottom two bytes are zero to conform to a standard absloc address usable in memloc(a,zmstart)

zmxnames 4(int) Maximum number of names permitted in the attached nameset type file

zname 8(alpha)First 8 characters of the selected name in the attached file; has no meaning for dataset files

zndisks 1(int) Number of drives defined within DOS as set by LASTDRIVE= in CONFIG.SYS

znindex 2(int) Position of the selected name within the attached nameset type file

znumber 2(int) Number of replacements made with -put

zoriginx 2(int) Relative x-coord of origin as set with -origin

zoriginy 2(int) Relative y-coord of origin as set with -origin

zpalincr 2(int) Minimum significant increment for R, G, and B values in -palette

zpcolo 2(int) Color of dot of last pointer input location; in text mode.foreground color of character

zplotxh 2(int) Upper x text extent (see text measure

zplotxl 2(int) Lower x text extent (see text measure

zplotyh 2(int) Upper y text extent (see text measure

zplotyl 2(int) Lower y text extent (see text measure

zrec 4(int) Number of records associated with the selected name in the attached file; for datasets equals "zfrecs"

zreturn 1(int) Indicates the success or failure of a command which sets this system variable. "zreturn" is generally set by commands which access disk or the TenCORE memory pool. Possible values for "zreturn" are:

BNSDOCID: <WO___9907007A2_I_>

2	Redundant operation (no action taken)
-1	Operation successful
0	Disk Error; see "zdoserr"
1	No file attached
2	Block out of range
3	Memory or value out of range
4	Does not exist
5	Invalid device selected
6	Duplicate name
7	Directory full
8	Insufficient disk records
9	No name in effect
10	Name or block not found11 Invalid type
12	Invalid name length
13	Invalid information length
14	Too many names/records in nameset
15	Nameset directory corrupted
16	Invalid name
17	Invalid image or data
18	Unable to fill memory pool requests
19	Operation invalid in context
20	datain- on locked data
21	Conflict with another user's lock

BNSDOCID: <WO_ _9907007A2_I_>

	22	Too many locks on one name	
	23	Screen coordinate out of range	
	24	Required name/record lock not found	

zrmargin 2(int) x-coordinate of right text margin

zrotate 2(int) Number of degrees of rotation

zrotatex 2(int) x-coordinate of rotation origin

zrotatey 2(int) y-coordinate of rotation origin

zrpage 1(int) Display read page set by -page

zrstartl 8(alpha)Lesson name of restart unit, set by -restart

zrstartu 8(alpha)Unit name of restart unit, set by -restart

zscaleox 2(int) x-coordinate of scaling origin

zscaleoy 2(int) y-coordinate of scaling origin

zscalex 8(real)x-scaling factor

zscaley 8(real)y-scaling factor

zscreen 1(int) Current hardware (BIOS) screen:

0, 1	40 x 25 text	cga,text,medium
4, 5	320x200 graph 4 color	cga,graphics,medium
6	640x200 graph 2 color	cga,graphics,high
7	80 x 25 mono text	mda
13	320x200 graph 16 color	ega,graphics,low
14	640x200 graph 16 color	ega,graphics,medium
16	640x350 graph 16 color	ega,graphics,high
17	640x480 graph 2 color	mcga,graphics,high
18	640x480 graph 16 color	vga,graphics,medium

19 320x200 graph 256 color mcga,graphics,medium

zscreenc 1(int) Current screen color type, as selected with the -screen- command: 0 = color; 1 = mono

zscreenh 1(int) Screen hardware driver (*.DIS file); adjusted for display hardware actually detected:

0 cga 10 vga

2 hercules 11 mcga

3 ega 4 evga

9 att

zscreenm 1(int) Current screen mode, as selected with the -screen- command: 0 = graphics, 1 = text

zscreenr 1(int) Current screen resolution, as selected with the -screen- command:

0 low 2 high

1 medium 3 alt1

4 alt2 5 alt3

zscreent 1(int) Current screen type, as selected with the -screen- command:

0	cga	5	ncr	10	vga
1	tecmar	6(reserved)		11	mcga
2	hercules	7(reserved)		12	online
3	ega	8	nokia	13	(reserved)
4	(reserved)	9	att	14	evga

zsdisks 4(bits)Bitmap of drives to include in automatic searches, set in DOS by SET TCSEARCH=

zserial 8(alpha)8-character serial number of TenCORE executor

BNSDOCID: <WO_ _ 9907007A2_I_>

zsetret	1(int)	Kind of match found when -setname- is executed; possible values are:
	N	Unique partial match (N-1 chars)
	-1	Exact match; a name is selected
	0	No match; selection cleared, no name selected
	+N	Non-unique partial match (N chars); first partially matching name selected
zspace	2(int)	Current character column
zsysver	4(alpha)	TenCORE version in the form "X.XX"
ztelmain	4(int)	Elapsed time spent in previous main unit
ztmmain	4(int)	System clock at start of main unit
ztmunit	4(int)	System clock at start of current unit
zuncover	2(int)	Current uncover key as set by uncover command
zunit	8alpha)	Unit name of current unit
zvarsl	4(int)	Total size of global variables, in bytes
zwidth	1(int)	Width of graphics lines set by -width-; affects draw, circle, ellipse, polygon, and dot
zwindows	2(int)	Number of windows opened (besides initial window
zwindowx	2(int)	Absolute x-coord of lower left corner of window
zwindowy	2(int)	Absolute y-coord of lower left corner of window
zwpage	1(int)	Display write page set by -page
zx	2(int)	Current graphic horizontal location
zxmax	2(int)	Maximum x-coordinate in current window
zxycolor	2(int)	Color of dot at current screen location
		In text mode, foreground color of character

BNSDOCID: <WO___9907007A2_I_>

zy	2(int)	Current graphic vertical location
zymax	2(int)	Maximum y-coordinate in current window

Displaying the Contents of Variables

Displaying Numeric Variables

Variables which contain numeric information can be displayed using the show command:

```
at      17:21
show    ztriea
```

The normal form is **show varname** where *varname* is the name of the variable to display.

For integers, **show** displays the first non-zero digit at the current screen location and displays up to 20 digits. For real numbers, it starts at the current screen location and displays up to 24 places, including a decimal point and three decimal places. This can be altered by adding *field* and *right* tags, as in:

```
show    real8,2,6
```

which displays up to a total of twelve places (including the decimal point): six digits including trailing zeros are displayed to the right of the decimal point. In this case, if the value of "real8" is 1.2345, it displays:

```
1.234500
(t      starting screen location)
```

Tabular Display

A second form of displaying numeric information, **showt**, right-justifies and space-fills the field when it displays a variable. This is useful for aligning columns of numbers. For example:

```
define  local
var1    ,8,r
var2    ,8,r
var3    ,8,r
```

```
var4      ,8,r
define    end
set       var1      <= 1.1, 20.02, 300.003, 4000.0004
at        5:5
showt     var1,12,4
at        6:5
showt     var2,12,4
at        7:5
showt     var3,12,4
at        8:5
showt     var4,12,4
```

The code above produces output similar to this:

```
1.1000
20.0200
300.0030
4000.0004
```

Hexadecimal Display

The **showh** command displays the contents of a variable using two hexadecimal base' 16 digits for each byte of the variable. For example:

```
calc      int4 <= 32766
showh     int4
```

displays:

```
00007ffe
```

SNSDOCID: <WO___9907007A2 I_>

Unless the optional length tag is specified, **showh** displays as many digits as necessary to show the entire variable (2 digits for a one-byte variable; 4 digits for a two-byte variable; etc.).

Alphanumeric Display

Text information is usually displayed using the **showa** (for "show alphanumeric") command.

```
packz    textbuf;;Good morning!
at       5:5
showa    textbuf        $$ displays Good morning"
```

The **showa** command displays characters for the length of the variable or for an optionally specified length.

```
showa    textbuf,6            $$ displays "Good m"
```

Embedded show

In practice, the **show** command and its variants are usually embedded in a **write** command, as in:

```
at       5:5
write    «show,real1»   «s,real1»   $$ 's' is for (s) how
         «showt,real1»  «t,real1»   $$ 't' for show (t)
         «showh,real1»  «h,real1»   $$ 'h' for show (h)
         «showa,text»   «a,text»    $$ 'a' for show(a)
```

Note that the abbreviations *s*, *t*, *h*, and *a* can also be used.

The characters « and » are called embed symbols. The left embedding symbol is produced by pressing [ALT][<], the right embed symbol is produced by pressing [ALT][>]

The general form for embedding a **show** command in a write is:

«command,expression»

Command is one of the **show** commands or its abbreviation; expression may be a number, a variable, or a calculation involving either, as in:

```
write    The Fahrenheit temperature is «s,9*celsius/5+32».
```

Assigning Values to Variables

Assigning values to variables is performed in various ways depending on the type of variable and the type of information to be stored in the variable.

Numeric Values

Values are assigned to numeric variables using the **calc** command:

```
define   local
ratio,4, real
define   end

calc     ratio ⇐ 1 / 3
```

The general form of **calc** is:

calc variable ⇐ expression

variable any author-defined numeric variable

⇐ the *assignment arrow* (produced by pressing [ALT][A] is the operator that assigns a value to the variable

expression a valid arithmetic and/or logical expression

In the example above, TenCORE calculates the value of 1 divided by 3 and stores it into the variable 'ratio'.

The **calc** command can be continued from one line to the next, as follows:

```
calc     a ⇐ 1
         b ⇐ 2
         c ⇐ b / 3
         d ⇐ c + b
```

BNSDOCID: <WO___9907007A2_I_>

The **calc** command above assigns values to four different variables without the necessity of repeating the command name on each line.

When a real value is assigned to an integer variable, the value is rounded to the nearest integer.

The **calc** command works only with valid numeric variables (i.e., 1,-2-, 3-, 4-, and 8-byte integers, and 4- and 8-byte reals). It does not work with 5-, 6-, or 7-byte variables, or with any variable larger than 8 bytes.

The full range of arithmetic and logical operators which can be used with **calc**, as well as alternate ways of representating numeric values, are treated in the section **Ways of Representing Literal Data** later in this chapter.

Selective Forms

The **calc** command has two selective forms, **calcc** and **calcs**. For more information, see the respective command descriptions.

Text Values

Because the **calc** command works only with numeric variables, i.e. variables with a defined length of 1, 2, 3, 4 or 8 bytes, it is not suitable for assigning text to variables.

Text is assigned to variables using the **packz** command. Its general form is:

packz buffer; [length]; text

buffer the buffer to receive the text

length an optional tag a variable to receive the number of charcters assigned to *variable* (the **packz** command sets *length*, not the author)

text the text information to assign to *variable*

The "z" on **packz** indicates that any bytes of *buffer* not used to store the text are zero filled. Without zeroing, packing "rose" into a buffer which already contained "geranium" would give "rosenium". If zero filling is not desired, use the **pack** command.

```
define   local
sublen,2
subject,40
define   end
*
packz    subject,sublen;Geography 101
*
at       5:5
write    The heading «a,subject» is «s,sublen» bytes long.
```

The *length* tag can be omitted, leaving two adjacent semicolons between the variable and the text to be assigned:

```
packz    subject;;Geography 101
*
at          5:5
write    The heading is «a,subject».
```

Like the **write** command, **packz** can be continued over several lines:

```
packz    text2;;This is line one.
         This is line two.
         This is line three.
```

Any of the **show** commands can be embedded in the text tag of the **packz** command just as they can be embedded in a **write** command:

```
packz    text;textlen;atomic weight of «a,element» is «s,weight».
at          5:5
show     text
```

BNSDOCID: <WO___9907007A2_I_>

When **showa** is embedded in a **packz** command, null characters(bytes with a value of 0) in the embedded variable are skipped over instead of being copied. If a special application requires that even the null characters be copied, the **showv** command can be used instead of **showa.** The **showv** command works just like **showa** with the important exception that even null characters are copied in a **packz** command or displayed in a **write** command.

The **showv** command has both an embedded form (abbreviated «v, ») and a non-embedded form, but is is primarily intended for embedding in a **packz** command.

Selective Form

The **packz** command has a selective form called **packzc**. For more information, see the respective command descriptions.

Copying Non-numeric data

The easiest way to copy non-numeric data from one variable to another is to use the **move** command, which has the general form:

move	source, destination, length	
	source	any defined variable or a literal
	destination	any defined variable
	length	an expression specifying the number of bytes to move.

The following example copies the contents of the 40-byte variable "newname" into the variable "oldname".

```
move        newname,oldname,40
```

Converting Text to Numbers

The **compute** command translates a string of numeric characters into a numeric value.

```
pack      text;text1;-12345
compute   text,text1,result
```

SNSDOCID: <WO 9907007A2_I_>

```
at        1:1
show      result
```

The general form is

compute buffer, length, result

buffer the string of characters

length the number of characters to convert

result the variable in which to store the result

The string to be convened can contain:

- the digits *0* through *9*
- the unary + and - characters (the positive or negative sign)
- one decimal point (period)

Any other characters within the string cause **compute** to fail, indicated by a **zreturn** value greater than -1. If it fails, **compute** makes no changes to the value stored in "result".

For extracting numeric information from the input at an **arrow**, see the **store** and **storen** command descriptions.

Initialization of Variables

Author-defined global variables should normally be initialized before use. This is often accomplished in a program's startup unit using the **zero** command.

zero

The simplest form of zero sets a single variable to zero, whatever its defined length:

```
zero      usernarne
```

Another form of **zero** clears a specified number of bytes starting at the named variable.

```
.zero     scores,22
```

Here, it zeros 22 bytes starting at "scores". The general form for clearing a number of bytes is:

BNSDOCID: <WO__9907007A2_I_>

zero variable,length

variable name of the variable at which to start

length the number of bytes to zero

Startup units often clear all of global variable space before assigning any values to variables. This ensures that the lesson has a clean slate to work with.

For clearing global variable space,. the system variable **zvarsl** gives the total number of bytes reserved for storage of global variables (including space which has not been defined). If "firstvar" is the first global variable defined, the following command zeroes all global variables:

```
zero    firstvar, zvarsl
```

If "firstvar" is not the first global variable defined, an area **zvarsl** bytes in length would extend beyond the end of global variable space. In this case, the **zero** command above results in an execution error.

Local variables do not need to be explicitly set to zero. All of a unit's local variables are automatically zeroed when execution of the unit begins.

set

Another way of initializing variables is with the **set** command. Rather than zeroing variables, set assigns a series of values to consecutive variables:

```
define  local
jan     ,1
feb     ,1
.                       $$ all twelve months
.
nov     ,1
dec     ,1
define end
```

```
set     jan ⇐ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
```

This example assigns 31 to "jan", 28 to "feb" and so on. It has the same effect as twelve **calc** commands.

The normal form of the tag is:

set variable ⇐ *value list*

variable the variable to assign the first value to.

value list the subsequent values that are assigned to locations following *variable*.

The **set** command does not perform any bounds checking: assignments are made based on the defined length of the variable named in the tag, regardless of how the subsequent variables are defined. In the above example, "jan" is defined as a 1-byte integer, so the values in the tag of the **set** are assigned to the twelve bytes starting at "jan". If the eleven variables defined immediately following "jan" are not all 1 byte long, they will not contain the expected values.

The **set** command has a selective form **setc**. For more information, see the respective command descriptions.

Comparison and Searching

Two variables can be compared using the **compare** command. This command returns the number of leading characters which match in the two comparison strings.

```
define   local
string1,26
string2,26
result,1
```

BNSDOCID: <WO___9907007A2 I >

```
define   end

pack     string1;;abcdefghijklmnopqrstuvwxyz
pack     string2;;abcDEFGHIJKLMNOPQRSTUVWXYZ
compare  string1,string2,26,result
```

This example compares "string1" to "string2" and sets "result" to the number of leading characters which matched; in this case 3, for 'abc'.

The general form of the compare command is:

compare var1,var2,length,result

var1 one of the variables to compare

var2 the other variable to compare

length the number of bytes to compare

result set to the position of the last matching byte. It is set to - 1 if *var1* and *var2* are identical or 0 if they do not match at all

find

Variables can be searched for specific contents using the **find** command.

```
define      local
alphabet,   26
aiphalen,   2
location,   2
packz    alphabet;alphalen;abcdefghijklmnopqrstuvwxyz

find        'ef',2,alphahet,alphalen,1,location
```

This example searches the variable 'alphabet" for the string 'ef. It finds it and sets the variable "location" to 5 because 'ef' starts at the fifth byte in the search area.

The general form of the **find** command is:

find object, object-length, start-var, list-len, incr, location

object the value to search for. This can be a literal, a variable, or a buffer

object-length the length of the object to search for, in bytes.

start-var the variable at which to start the search

list-len the number of entries to search. An entry can be more than one byte long, as specified by *incr.*

incr the length of each entry.

location the entry number at which *object* was found If *object* was not found, *location* is set to -1.

Examples

The following examples assume definition of the following variables:

```
define      local
name        ,15    $$ name to search for
found       ,2     $$ position in list where "name" found
list(5)     , 15   $$ list of names
define             end
```

The variable "list" contains a list of 5 names, each of which occupies 15 characters. The contents of "list" are as follows:

john miller... mark ho.....sarah johnston.james heflin lisa berger

↑1 ↑16 ↑31 ↑46 ↑61

Null characters have been shown as to improve their visibility, and numbers have been added to help in counting bytes.

Literal Object

The following **find** statement searches for the name *mark ho:*

```
find      'mark ho',7,list(1),5,15,found          $$ found will be 2
```

Because the name has fewer than 9 characters, it can be supplied as the text literal 'mark ho'. Next comes the length of the object to find, 7 characters. The start of the list is given as list*(1)*, and its length as 5 entries of 15 bytes each.

After the search, the variable "found" contains the value 2 because *mark ho* is found at the second position (not the second byte) in the list.

Variable Object

Since names of more than 8 bytes cannot occur as text literals, they must be packed into variables. The following example locates *james heflin.*

```
packz    name;;james heflin
find     name,15,list(1),5,15,found
```

The variable "found" receives the value 4 because *james heflin* is found in the fourth position.

Byte-by-byte

To find a last name, a **find** like the following could be executed:

```
packz    name;;miller
find     name,6,list(1),75,1,found
```

When this example is executed, "found" receives the value 6.

The length of the name is given as 6 because this locates part of an entry, not an entire entry.

Similarly, the list is specified as 75 one-byte entries instead of 5 fifteen-byte entries. This causes **find** to look for *miller* starting at every character, not just at the start of each 15-byte name field. This illustrates that the object of the search can be longer than the nominal entry length provided to find.

Backwards by Entry

To search backwards, a negative value is given for the increment. The following example searches backwards from the end of the list for *john*, looking only at the beginning of each 15-byte entry:

```
find     'john',4,list(1),5,-15,found
```

Here, "found" receives the value 1. Although the search proceeded backwards from the end of the list, the position is always counted from the beginning of the list.

Backwards by Byte

Another example of backwards searching is the following:

```
find     'john',4,list(1),75,-1,found
```

In this case, an increment of-1 is used, causing find to look at every character starting from the end of the list.

In this case, "found" receives the value 37, because the first *john* found when searching backwards byte-by-byte occurs in *sarah johnston*. If the search had been in the forward direction. *john miller* would have been found first, and "found" would have received the value 1.

Ways of Representing Literal Data

Data used in **calc** commands and other numeric contexts can be expressed in a number of literal forms.

For purposes of discussion, the term *literal* is used to identify a 1-, 2-, 3-, 4- or 8-byte value which is expressed directly rather than as a variable or an expression. Values of 5, 6 or 7 bytes can be used as literals, but are represented internally as 8-byte values.

Integer Literals

Any valid number written as a sequence of digits with no decimal point is an integer literal. An integer literal may have a leading plus or minus sign.

Real Literals

A real literal is distinguished from an integer literal by the presence of a decimal point. Even if there are no digits following the decimal point, the presence of the decimal point causes the value to be represented as a real value.

Text Literals

A text literal is a string of 1 to 8 characters enclosed in single quotation marks. Many commands allow file names, unit names or other block names to be written as text literals:

```
image    plot;'pictures','bird'
```

The **image** command above displays a screen image stored in file pictures, block **bird.**

Text literals are always stored internally as 8 bytes; unused bytes at the right end of the text are zeroed. Another way of stating this is that text literals are stored *left-justified* in 8 bytes.

Because text literals are 8 bytes long, the **calc** command can be used to assign a text literal to an 8-byte variable:

```
calc     filename ⇐     'pictures'
```

The variable must have been defined to be 8 bytes long. If a shorter variable is used, characters are certain to be lost from the *beginning* of the text literal.

Character Literals

Character literals normally consist of a single character enclosed in double quotation marks:

```
if       name(2) = "a"
```

Such a character literal is stored internally as a single byte.

A character literal can be assigned directly to an integer-compatible variable using **calc:**

```
calc     name(2) ⇐ "z"
```

It is also possible, although unusual, to create multiple-byte character literals by enclosing a string of 2 to 8 characters in double quotes:

```
calc      chars ⇐ "abc"
```

Such a literal is always stored in the smallest possible of 1, 2, 3, 4 or 8 bytes. If the literal contains 5, 6 or 7 characters, zero-valued bytes are added to the *left* end of the literal to round its length up to 8 bytes. Another way of saying this is that character literals are stored *right-justified* in the smallest possible size of 1, 2, 3, 4 or 8 bytes.

Multi-byte character literals (enclosed in double quotes) are not a substitute for text literals (enclosed in single quotes). Correct usage is to use single quotes for all textual literals such as file names and unit names, and to reserve double quotes for single characters.

Keypress Literals

When examining the system-defined variable **zinput,** which reports the last key processed, one often uses *keypress literals.*

A keypress literal is normally a 1-byte character literal prefixed with a percent sign as follows: %"a". The percent sign changes the 1-byte character literal into a 2-byte internal representation consistent with **zinput**.

A number of other prefixes exist as well: %alt, %ctl, %altctl and %ctlalt (the last two are equivalent). Thus, to determine whether the last key processed was [CTRL][A] one could write

```
if zinput = %ctl"a"
```

All keypress literals are represented internally as two byte positive values. The detailed format can be found under the description of **zinput**.

Hexadecimal Literals

This discussion of hexadecimal (base 16) literals should be considered optional advanced material. If you have not used hexadecimal numbers before, you probably do not need this information and you may want to skip ahead to *Constants.*

BNSDOCID: <WO__9907007A2 I >

A hexadecimal literal is written as the letter *h* followed by a digit 0 - 9, followed by zero or more of the hexadecimal digits 0 - 9 and *a* - *f* (In hexadecimal notation, the values 10 through 15 are represented by the leners *a* through *f*.)

Thus the following are valid hexadecimal literals:

```
h4        (equals decimal       4)
h10       (equals decimal      16)
h0f       (equals decimal      15)
h1ec4     (equals decimal    7876)
hffff     (equals decimal      -1)
```

Note that *hf* would not be a valid hexadecimal literal, since the first digit after *h* must be within the range 0 - 9. Any time the first digit of the hexadecimal value is one of *a* through *f*, an extra 0 must appear after the *h* as in: *h0f*.

It is useful to think of hexadecimal literals as being evaluated "backwards", from right to left. The digits of the literal are read in pairs, starting from the right, and each pair of digits corresponds to one byte in the internal representation of the value. Each new pair of digits causes a new byte to be added to the internal representation of the literal. A lone, unpaired digit remaining at the left end of the literal also causes another byte to be added, *unless the remaining digit is zero.* A lone, unpaired zero at the left end of the literal is always discarded. If the resulting internal representation is 5, 6 or 7 bytes long, zero-valued bytes are added to the left of the value to pad it out to 8 bytes.

Hexadecimal Literals as Integer Values

In calculations, a hexadecimal literal is always considered an integer value,.When assigning a hexadecimal literal to a variable, it is important to distinguish between the length of the variable and the length of the literal. Consider the following example:

```
define      local
```

```
temp,4
define      end
*
calc      temp <= h91
```

Here the value of a 1-byte literal is assigned to a 4-byte integer variable. The integer value of the literal is evaluated before considering the length of the variable. Since the literal is one byte long, its value is h91, which corresponds to the binary value 10010001. Because the leftmost bit is a '1', the value is interpreted as a negative number: 111. This is the value assigned to the variable "temp". Note that the two-byte literal h0091 or the four-byte literal h00000091 would have given a different result, namely the positive number 145. This is why it is important to pay attention to the length of a hexadecimal literal.

Hexadecimal Literals as Bit Patterns

It is sometimes desirable to use a hexadecimal literal to assign a particular pattern of bits to a real variable. This cannot be done using **calc**, because a calculation would interpret the hexadecimal literal as an integer, and then convert the value of the integer to the IEEE format used to store real numbers in TenCORE. The numeric value would be preserved, but the bit pattern would not.

The **move** command can be used to copy a hexadecimal literal into a variable as a bit pattern without any numeric interpretation:

```
define      local
height,4,   real
define      end
*
move        h7f800000,height,4
```

This example copies the 4-byte bit pattern for the indefinite value 1/0 into the 4-byte real variable "height". Because no calculation is performed, no conversion of the bit pattern takes place.

Constants

A constant is a named item of literal data. Constants can be defined globally or locally using the = sign:

```
a=1
greet='Hello'
```

The general syntax for constant definition is *name=literal.* A constant can be given any name which is valid for a variable. Constant definitions can occur at any point where a variable definition can occur. Once defined, a constant can be used anywhere that the literal data it represents could be used:

```
define      local
Picfile='pictures'
picblock='fish'
Px=140
Py=100
define end
*
image    plot;block,Picfile,Picblock;Px,Py
```

The **image** command above is equivalent to:

```
image    plot;block,'pictures','fish';140,100
```

Constants are often defined for use in defining other variables:

```
MAXNAMES=20
name(MAXNAMES),20
age(MAXNAMES),2
```

```
address(MAXNAMES),40
```

This example defines three arrays of length 20. The lengths of all three arrays can be changed by changing the definition of MAXNAMES.

Many programmers make a convention of using upper-case characters for names of constants so they are not easily confused with variable names.

The same rules apply to literals used in constant definitions as to literals used anywhere else. A constant has a value, a type (integer or real), and a length (the number of bytes in the internal representation). As with other literal data, all three of these characteristics must sometimes be considered when using a constant in a calculation.

Operators

TenCORE supports a rich variety of operators which can be used to combine literals, constants and variables into numeric expressions. In addition, TenCORE supports a number of mathematical functions and provides a mechanism for the author to define other functions.

Arithmetic Operators

+	Addition
⇐	Assignment (e.g., a ⇐ b means assign b value to a) [CTRL][C][A] or [ALT][A]
–	Subtraction
* or x	Multiplication, real arithmetic [CTRL][C][+]
Simul$	Multiplication. integer arithmefic
or÷	Division, real arithmetic {CTRL][C][/]
$idiv$	Division, integer arithmetic
**	Exponent (e.g., a ** b means a to the b power)
°	Convert preceding value from degrees to radians [CTRL][C][°]

BNSDOCID: <WO _9907007A2 I >

Logical Operators

Logical operators compare expressions and return values corresponding to true or false. For example, the command sequence:

```
if      2 <     3
.       write   True
endif
```

plots True because 2 is less than 3. This is similar to posing the question:

True or false: Two is less than three.

Logical operators compare the operands in an expression and then return value - 1 if the expression is true, 0 if false. Key sequences used to enter non-standard characters are shown.

=	Equal to (true if operands are idenfical)
≠	Not equal to (true if operands are not identical)
<	Less than (true if leti operand is less than right operand)
>	Greater than (true if left operand is greater than right operand)
≤	Less than or equal to (true if left operand is less than or equal to right operand)
≥	Greater than or equal to (true if left operand is greater than or equal to right operand)
and	Logical "and" (true if both operands are true
or	Logical "or" (true if either operand is true)

In addition to the operators above, the **not()** function returns true if the operand in parentheses is false and vice versa.

The operands of a logical operator do not have to be numeric values; for example, ASCII characters can be compared with each other.

In TenCORE, true is represented by a value of -1 and false is represented by a value of 0. For example, some expressions and their numeric values are:

```
2        <  3      $$ returns -1, true
2        =  3      $$ returns 0, false
2        >  3      $$ returns 0, false
not( 2 > 3 )       $$ returns -1, true (true is the same as not false)
```

More generally, TenCORE treats any negative value as true and any non-negative value as false.

Because true and false have associated numeric values, a logical expression is often used as the selector in the selective form of a command:

```
writec   2<3;True;False
```

This example prints *True* because 2 is less than 3 and the result of the expression is thus -1; had the expression been false, the numeric value would have been 0 and the example would have printed *False*.

Although logical operators are normally used in combination with each other, they are occasionally used together with arithmetic operators as in the following:

```
calc        score <= score - (annwer=correct)
*           increments "score" by 1 if answer=correct
```

This has the effect of leaving "score" unchanged if the values of "answer" and "correct" are different. This is because the value *of answer=correct* will be False (0), and the value assigned to "score" is *score - (0)*. But if the values of "answer" and "correct" are equal, then answer=correct has the value True (-1) and the value assigned to "score" is *score - (-1)*, which is equivalent to *score + 1*.

Bitwise Operators

Bitwise operators treat each operand as a pattern of bits rather than as a numeric value.

The bitwise operators and the operations they perform are:

$mask$ Logical "and". Each bit in the result is set if the corresponding bit is set in both operands

union$ Logical "or". Each bit in the result is set if the corresponding bit is set in either operand.

$diff$ Logical "exclusive or". Each bit in the result is set if the corresponding bit is set in one of the operands and not set in the other operand.

ars Arithmetic Right Shift. The bit pattern in the first operand is shifted right by the number of places given in the second operand. The leftmost (sign) bit propagates to the right, and bits which "fall off" the right end are discarded.

$cls1$ Circular Left Shift in 1 byte. The bit pattern in the first operand is circularly shifted leftwards for the number of places given in the second operand. Bits shifted out of the left end of the byte re-appear in the right end.

$cls2$ Circular Left Shift in 2 bytes.

$cls3$ Circular Left Shift in 3 bytes.

$cls4$ Circular Left Shift in 4 bytes.

$cls8$ Circular Left Shift in 8 bytes.

For example, $mask$, $union$ and $diff$ produce the following results:

```
          $mask$   $union$     $diff$
45        00101101 00101101    00101101
84        01010100 01010100    01010100
          -------- --------    --------
```

ENSDOCID <WO 9907007A2 I >

```
121       00000100 01111101    01111001
```

Thus 45 $mask$ 84 equals 4; 45 $union$ 84 equals 125; and 45 $diff$ 84 equals 121.

Bitwise operators are intended to operate only on integer values and non-numeric values with valid integer sizes (1 ,2 ,3 ,4 or 8 bytes). They are not suited for use with real values.

System-Defined Functions

TenCORE supports a number of system-defined functions which take an argument in parentheses. These include trigonometric, logarithmic, logical, arithmetic and address functions.

Logarithmic

alog(x)	Antilogarithm base 10 of x (10 to the x^{th} power)
exp(x)	Math constant C to the x^{th} power
ln(x)	Natural logarithm of x
log(x)	Logarithm base 10 of x
logx(n,x)	Logarithm base n of x

Negation

comp(x)	Bitwise complementation. All bits in operand x are inverted (bits containing '1' are reset to 0' and vice versa).
not(x)	Logical negation. Returns true (-1) if operand xis false and vice versa.

Trigonometric

acos(x)	Arc cosine (x in radians)
asin(x)	Arc sine (x in radians)
atan(x)	Arc tangent (x in radians)
cos(x)	Cosine (x in radians)
sin(x)	Sine (x in radians)
tan(x)	Tangent (x in radians)

Address

absloc(v)	4-byte absolute address in memory of variable v
sysloc(global)	Segment address of global variable space
sysloc(local)	Segment address of local variable space for current unit
varloc(v)	Offset from start of local or global variable space for variable v

Arithmetic

abs(x)	Absolute value of x
bit(v,x)	Logical value of bit x in variable v (-1 if the bit is 1)
bitcnt(v,n)	Number of bits set in variable or buffer V, in a range of n bits
frac(x)	Fractional part of x
imod(x,y)	x modulo y
int(x)	Integer part of variable x
mod(x,y)	x modulo y
randi(x)	Random integer from 1 to x
randi(x,y)	Random integer from x to y
randr(x)	Random real value from 0.0 to x
randr(x,y)	Random real value from x to y
round(x)	x rounded to the closest even integer
sign(x)	-1 if x is negative
	0 if variable x is positive
sqrt(x)	Square root of x
π	Pi(3.14159265358979312) [CTRL][C][P]

Evaluation of Expressions

This section discusses the details of how TenCORE evaluates expressions. The first part, Precedence of Operators, is useful to all authors. The remainder of the material can be regarded as optional information for authors who need to optimize calculations for maximum possible efficiency of execution.

Precedence of Operators and Functions

TenCORE observes a certain precedence of operators in evaluating expressions. Thus the expression 5 + 4 * 3 equals 17 instead of 27 because the multiplication operator * has a higher precedence than the addition operator +, and is executed first. This makes the expression 5 + 12 instead of 9 * 3

Parentheses override the normal order; operations within the parentheses are done first (in normal order). The resulting value is then used in the related operation in the expression. The following table shows the TenCORE operators and functions in the order in which they are performed. Class 1 is performed before class 2, 2 before 3, and so on.

1	Exponentiation (**) and Functions
2	Unary Minus and Unary Plus
3	Multiplication and Division (*, /, ÷, $imul$, $idiv$)
4	Addition and Subtraction (+, -)
5	Bitwise Operators ($union$, $mask$, $diff$, ars, $cls1$, $cls2$, $cls3$, $cls4$, $cls8$)
6	Comparison Operators (=, ≠, <,>, ≤ ,≥)
7	Logical Operators (and, or)
8	Degree-to-Radian Conversion (° as in 45°)

9 Assignment (⇐)

Rounding

Rounding of a value with a fractional part exactly equal to .5 always yields an even integer Another way of stating this is that values where the fractional part is .5:

- round up if the integer part is odd
- round down if the integer part is even

Example

```
calc        intvar ⇐ 7.5            $$ "intvar" assigned the value 8
            intvar ⇐ 8.5            $$ "intvar" assigned the value 8
```

Compression of Sub-Expressions

Some sub-expressions are evaluated and compressed when a unit is translated to binary form rather than when it is executed:

```
calc      ratio ⇐ (4 + 5) / limit
```

Because the sub-expression (4+5) contains only integers and its value is known already before the program is run, the command would be represented internally as:

```
calc      ratio ⇐ 9 / limit
```

To qualify for evaluation and compression at translation time, a sub-expression must contain only:

- integer literals which fit in 4 bytes or less
- integer constants which fit in 4 bytes or less
- the operators + - $imuls $idiv$

Sub-expressions containing other elements cannot be evaluated at condense time and are not compressed.

Operator Types

Certain operators and functions work exclusively on a particular type of value, either real or integer. Non-conforming values are internally converted into the type expected by the operator or function before any operation is performed.

Operators which work with real values are:

*	real multiplication
/	real division
**	exponentiation
°	degree-to-radian conversion

In addition, the trigonometric functions, exponential and logarithmic functions, and int(), frac() and round() always work with real values.

When these operators and functions are used with integer operands, the operands are first converted to real values. If the result is assigned to an integer variable or used in a command which expects an integer value, the result is convened to back to integer.

Operators which work only with integer values are:

$imul$	integer multiplication
$idiv$	integer division

In addition, the logical and bitwise operators, and the functions imod(), not() and comp(), always work with integer values.

When these operators and functions are used with real operands, the operands are first converted to integer values. If the result is assigned to a real variable, it is converted to real before assignment.

Other operators and functions can work on real or integer values and do not perform any internal conversions.

BNSDOCID: <WO__ 9907007A2_I_>

Operator types normally affect only a program's speed of execution. However, certain integer operators applied to real values may give unexpected results if the implicit conversion is not kept in mind. The bitwise operators in particular, together with the comp() function, don't really give meaningful results when applied to real values.

Intermediate Values and Speed Optimization

A TenCORE author usually does not need to know how intermediate values used in evaluating expressions are represented internally, as the internal representation does not affect the correctness of the result. However, the information can be useful in optimizing calculations for speed.

Intermediate values are always represented as either 4-byte integers or extended-precision 10-byte reals, Four-byte integers are used if all three of the following conditions are met:

- none of the operands are real
- none of the operators require real values
- none of the operands are longer than 4 bytes

If any of the above conditions are not met, 10-byte reals are used. Since 10-byte real calculations take several times as long as 4-byte integer calculations, a single real operand in an otherwise (4-byte) integer expression can significantly slow down its evaluation. In practice, this is usually significant only in particularly speed-critical applications, especially if a math coprocesor is not present.

Author- Defined Functions

An author-defined function is a named expression. Like constants, functions are defined using the = sign:

```
pressure = weight / area
```

The simplest syntax for function definition is *name=expression.* A function can have any name which is valid for a variable. Function definitions can occur at any point where a variable definition can occur. Once defined, a function can be used anywhere that the expression it represents could be used.

Arguments

Functions can be defined with arguments:

```
fahr(xx) = 9 * xx / 5 + 32

inrange(aa,bb,cc) = aa ≤ bb $and$ bb ≤ cc
```

The function "fahr()" defined above gives the Fahrenheit equivalent of a Celsius temperature. It could be used as follows:

```
write    15° celsius is «s,fahr(15)» Fahrenheit
```

The function "inrange()" determines whether one value lies between two others, returning true or false. It could be used as follows:

```
if          inrange(100,zinputx,200) $and$ inrange(140,zinputy,180)
.           write Good! You pointed inside the box.
endif
```

This example checks whether the last pointer input (given by the system variables zinputx,zinputy) was within a rectangular area with opposite corners at 100,140 and 200,180.

The general syntax for defining a function with arguments is

name(*arg1*,*arg2*,...,argn)=*expression*

An argument can have any name which is valid for a variable, but must not duplicate the name of a variable which is already defined. Many authors use double letters such as aa, bb, etc., to minimize the likelihood of using an argument name which conflicts with a variable or constant name.

BNSDOCID: <WO__ 9907007A2_I_>

The purpose of author-defined functions is to improve readability of source code and to simplify changes. When a program is translated to binary form, each reference to a function is expanded to the full length of the definition. Using functions thus has no effect on the final size or efficiency of a program.

Reduction to Constants

Any integer-valued function which can be entirely evaluated and compressed at condense time is reduced to a constant (see the Compression of Sub-Expressions section earlier in this chapter). Thus the following two definitions are equivalent:

```
length = 4 + 256 + 128
```

```
length = 388
```

Reduction of simple functions to constants is particularly useful when defining variables, as it makes possible constructions like the following:

```
define   local
HEADER   = 4
ENTRIES  = 20
LENGTH   = HEADER + ENTRIES
list(LENGTH),1
define   end
```

Type and Length

Like literals, constants, and variables, functions have an associated type and length. These are determined according to the rules given above for evaluating expressions. Thus the value of a function is always either a 4-byte integer or an 8-byte real. In practice, this is seldom significant, since TenCORE automatically performs any needed type conversions.

Physical Allocation of Variable Space

In keeping with the differences in characteristics and usage between global and local variables discussed in chapter 4, they are stored in different physical segments of the computer 8 memory.

Global Variable Storage

Global variables occupy a segment of memory which is reserved when TenCORE is started and is preserved until TenCORE is exited. There is only one segment of memory used for global storage. Every defines block and every **define global** statement allocates storage for variables starting at the beginning of this single segment.

The system-defined variable **zvarsl** gives the length of the memory segment allotted for global variables.

Local Variable Storage

Local variables occupy a segment of memory specific to the unit in which they are defined. Each unit has its own separate local storage segment which exists only as long is the unit is active. This segment is reserved and zeroed when execution of the unit begins. When the unit is exited by branching to a new main unit, by reaching the end of the unit, or by exit via **return** or **goto q**), the space for that unit's local variables is released.

Allocation of Space

Within a set of global or local defines, successive bytes of storage are allocated sequentially starting from the beginning of the global or local segment. For example:

Defintion	Location(offset from start of segment)
`var1,2`	Bytes 0 - 1
`var2,2`	Bytes 2 - 3
`realvar,8,r`	Bytes 4 - 11
`userrame,40`	Bytes 12 - 51

Two system-defined functions return the location of a variable within memory:

BNSDOCID: <WO_ 9907007A2 I >

varloc() returns the offset within variable space (local or global)

absloc() returns the absolute location within computer memory

The starting byte of each variable above could be displayed as follows:

```
at        5:5
write     var1      starts at byte «s,varloc(var1)»
          var2      starts at byte «s,varloc(var2)»
          realvar starts at byte «s,varloc(realvar)»
          username starts at byte «s,varloc(buffer)»
```

Bytes of storage can be skipped, as in:

```
var1,2
,4
var2,2
```

where ,4 allocates four bytes as unused.

Absolute Definition

Variables can be defined at an absolute offset (from the beginning of local or global space, as appropriate) by using the format:

@offset, name, length, type

where the @ symbol indicates that the variable is to start at the specified location. For example:

```
@0,first,1          $$ first byte of variable space
@1000,name, 8, i    $$ 8-byte integer at offset 1000
```

The variable "first" above could be used with **zero** to zero all global variables as follows:

```
zero      first,zvars1
```

By defining "first" at an absolute location (rather than just placing it first in the defines block), the author guarantees that the **zero** command will always start with the first byte of global

storage, even if other variable definitions are inadvertently inserted before "first" in the defines block.

The offset for an absolutely defined variable may be given as a defined constant:

```
loc = 2000              $$ defined constant
@loc,xyz,r              $$"xyz" starts at offset 2000
```

The use of absolute definition does not affect the allocation of subsequent variables--allocation continues sequentially following the last variable which did not use absolute variable definition.

Redefinition

A technique very similar to absolute definition is redefinition, in which a variable is defined to start at the same location as a previously defined variable. Redefinition takes the format:

@oldname, newname, bytes, type

where the @ symbol indicates that the variable newname is to start at the same location as *oldname*. For example:

```
username,40             $$ full user name
@username,userchar,1    $$ first character redefined/1-byte integer
```

In this example, the variable "usercha" could be used with if to test whether the first character (and thus presumably the whole name) is null:

```
if      userchar = 0
.       do getname
endif
```

As in the case of absolute definition, redefinition does not affect the allocation of subsequent variables: allocation continues sequentially following the last variable which did not use redefinition.

Segmentation

Variables may be broken down into smaller pieces or segments which can be referenced by name.

```
define   local
systime ,4
.           hours       ,1
.           minutes     ,1
.           seconds     ,1
.           hundr       ,1
define end


clock    systime
at       3:3
write    time: «s,hours» hrs, «s,minutes» min, «s,aeconds» sec
```

With "systime" segmented as above, the bytes can be referenced collectively (as in the **clock** command) or individually (as in the **write** command).

The definition of a segment must follow the definition of the variable of which it is a part. A period must appear in the first position, then seven space characters (one tab), then the segment name and size.

Segmentation is sometimes used in defining buffers for disk operations, which always use 256-byte lengths of data.

```
record,256
.         name,40
.         response(64),1
.         score,4
time,4
```

Here, "record" reserves a full 256 bytes, even though the segments only need 40 + 64 + 4, or 108, bytes. This ensures that that one can read in a 256-byte disk record without affecting "time" or other neighboring variables.

As this example illustrates, segments do not have to add up to the full size of the segmented variable. However, they normally must not exceed it.

Only one level of indentation is allowed when defining segmented variables. If a segment needs to be further divided into sub-segments, this can be accomplished by combining segmentation and redefinition:

```
userid,20
.           u.id1,10
.           u.div,1
.           u.id2,21
*
@u.id1,,10                          $$ redefinition of u.id1
.           u.id1.a,1
.           u.id1.b,9
```

No name was given to the redefined variable. It could have been given a unique name such as "xuid1", but since the only purpose was to (sub-)segment "u.id1", it was simpler to omit the name.

The use of period (.) in variable names has no special meaning in TenCORE, which treats the period just like any other valid character, but does give visual emphasis to the logical structure.

Array Segmentation

Segmentation of arrays is similar to segmentation of individual *(scalar)* variables. However, when an array is segmented, each segment is itself an array, even though no array size is given.

```
define      local
systime(3)              ,4
.           hours       ,1
.           minutes     ,1
.           seconds     ,1
.           hundr       ,1
define end
*
clock       systime(1)
clock       systime(2)
clock       systime(3)
*
write       The three times are:

        «s,hours(1)» hrs, «s,minutes(1)» min, «s,seconds(1)» seconds
        «s.hours(2)» hrs, «s,minutes(2)» min, «s,seconds(2)» seconds
        «s,hours(3)» hrs, «s,minutes(3)» min, «s,seconds(3)» seconds
```

Because the variable "systime(3)" was defined as an array with 3 elements, each of the segments "hours", "minutes", "seconds" and "hundr" is treated as an array which is referenced with an index in parentheses.

Special Segmentation

In certain cases, especially with anays, it is desirable to segment beyond the defined length of a variable. Normally, this would result in an error when the program is compiled before execution. However, by adding the keyword **special** (abbreviated **s**) after a segment definition, the check for segmenting beyond the length of the original variable is omitted. The defaults for variable definitions cannot be used with this keyword: the length and type must be explicitly specified, with the keyword **special** added after the type.

Segmentation beyond the end of a variable does not cause any more bytes to be allocated in variables. After the special segments are defined, the next variable starts at the location after the last normally defined variable.

Example

Shows how one could define a data area containing mixed data types: two-byte integers and eight-byte integers. Each piece of data is prefixed by a single byte indicating which type of data follows.

```
data,8192
@data,buf1(8192),1
.        type1,1,i,s    $$ prefix byte
.        data1,2,i,s    $$ access 2-byte words at any byte
@data,buf2 (8192),1
.        type2,1,i,s    $$ prefix byte
.        data2,8,i,s    $$ access 8-byte words at any byte
newvar,1 $$ starts 8192 bytes after "data"
            $$ newar is overwritten by special segments
```

By knowing the starting byte of an individual entry, the type can be determined and the appropriate variable name ("data1" or "data2") used to access the data in the appropriate format. The next entry would be found by incrementing the index into the array by the length of the current entry (3 or 9).

Blocks

A technique similar to segmentation but applicable on a larger scale is the variable block. Defining a variable block reserves a section of variable space. Within the variable block, scalar and array variables can be defined and segmented.

The normal format for defining a variable block is:

name, size, **block**

where name is any name valid for a variable; size is the number of bytes to reserve for the block; and **block** indicates that this is a variable block (**block** can be abbreviated as **b**).

The following example defines a 256-byte block named "diskio":

```
diskio,256,block  $$ 256-byte data area for disk input/output
,10       $$ 10 bytes unused
name,20   $$ user's name
date,6    $$ date of last use
time,4    $$ time of last use
.         hours     ,1
.         minutes   ,1
.         seconds   ,1
.         hund      ,1
score,2   $$ most recent test score
status,256,block  $$ next block begins here
```

Once a variable block is defined, all subsequent variable definitions are part of the defined block, up to the next variable block definition. If definitions within a variable block use up more space than is reserved for the block, a compile error occurs.

When the length of the variable block is not known, the *size* argument can be left blank. In this case, the variable block is given whatever size is required to hold all variables subsequently defined, up until the next block definition.

Sharing Definitions Among Source Files

If a program is spread over several source files, care must be taken to make the global variable definitions agree for each file. This is normally done by placing all the global definitions in one file and including a copy of the definitions in the other files with use commands:

3NSDOCID: <WO___9907007A2_I_>

tax 1,globals (defines block)

```
lastname,15
firstnam,10
mi,1
stunum,9
p(64),1
score,4
```

tax 1a,globals (defines block)

```
use     tax1,globals
```

tax 1b,globals (defines block)

```
use     tax1,globals
```

In this example, all of the variables for the files **tax1, tax1a** and **tax1b** are defined in **tax1**. The use commands in **tax1a** and **tax1b** just include a copy of the definitions already made in **tax1.**

Sharing Some Definitions But Not Others

Sometimes, it is desirable to define some variables which are used by an entire set of source files, and other variables which are unique to each individual source file. Since there is only one memory area for all global variables, the definitions of the variables for individual source files must naturally be coordinated to avoid conflicting definitions. This can be accomplished using a combination of variable blocks and redefinition.

The following example applies this technique to the files **tax1, tax1a** and **tax1b** by adding variables defined in **tax1a** and **tax1b:**

tax1,globals (defines block)

```
lastname,15
firstnam,10
mi,1
stunum,9
p(64),1
score,4
*
tax1a,2048,block
tax1b,2048,block
```

tax1a,globals (defines block)

```
use          tax1,globals
*
@tax1a,ownvars,2048,block
*
xloc,2
yloc,2
interest,8,r
```

tax1b,globals (defines block)

```
use       tax1,globals
*
@tax1b,ownvars, 2048,block
line11a,8,r
line11b,8,r
line12,8,r
```

In this example, the definitions used by all three files include the 2048-byte variable blocks "tax1a" and "tax lb". These variable blocks are data areas reserved for the source files of the same name. The source files **tax1a** and **tax1b** then redefine their own block using the name "ownvars" and define their own variables within this block. Making "ownvars" a block variable ensures that the variable definitions in **tax1a** and **tax1b** do not extend beyond the areas reserved for them.

Unlike other redefined variables, redefined block variables permanently affect variable allocation. If **tax1a** defines another block after "ownvars", this block will conflict with the variables defined in **tax1b**.

Accessing Other Memory Segments

Sometimes it is desirable to access data stored in areas of memory other than local and global variable space. This can be accomplished using the **transfr** and **exchang** commands.

transfr

The **transfr** command transfers data from one place to another. It has the general form:

transfr from-base-address,offset;to-base-address,offset;length

Both *from-base* and *to-base* must be from among the following keywords:

- **routvars, r**

  Router variables. This is a special unstructured 256-byte buffer which can be accessed only via **transfr** and **exechang.** The router buffer provides an area for data that is not normally used by lessons. One common use for this buffer is for keeping user performance data for an activity manager.

- **display, d**

  CGA screen display memory. Note that display memory for other display types EGA, VGA, etc.) cannot be accessed with **transfr**.

- **global,g**

ENSDOCID: <WO 9907007A2 I >

Global variables.

- **local,l**

  Local variables.

- **sysvars, s**

  System data area. **CAUTION:** do not use this area for data storage.

- **sysprog,p**

  System program area. **CAUTION:** do not use this area for data storage.

- **absolute, a**

  Absolute memory location.

The following example transfers the router variables into local variables and then displays the student's sign-on name.

```
define      local
localr,256
.           info,8
.           name,20
define      end


transfr  routvars,0; local, varloc(localr);256
at       10:5
write       The student name is «a,name»
```

exchang

The **exchang** command is similar to the **transfr** command, but swaps two areas of memory.

The following code uses the **exchang** command to swap a pair of 2-byte variables:

```
if      xl < x0
.       exchang g,varloc(x0);g,varloc(xl);2
endif
```

This is equivalent to but simpler than the following:

```
        if      xl  < x0
.       calc    temp  ⇐ x0
                x0    ⇐ xl
.          xl   ⇐ temp
endif
```

Accessing Absolute Memory

The **transfr** and **exchang** commands can also be used to access any absolute memory address in the computer. This is accomplished using the base-address keyword **absolute**. The offset is then a four-byte integer location.

A useful function when transferring to or from absolute memory locations is absloc(), which returns the absolute location in memory of a variable. This is often used to pass the location of data to a library routine, as follows:

```
pack    name;namelen;Wendell Oliver Holmes
library mclib,answer(absloc(name),namelen)
```

The advantage to using absloc() here is that unit **mclib,answer** does not need to know whether the data is in the local or global segment--it can copy the data directly from the absolute memory location.

Technical Notes on Absolute Addresses

The absloc() function returns four-byte values over a continuous integer range. To convert an absloc() value to the segment:offset form required for passing data to DOS functions, the following calculations can be used:

```
define      local
temp        ,4
segment     ,2
offset      ,2
define      end


calc        temp      <= absloc(myvar)
            segment   <= temp $ars$ 4
            offset    <= temp $mask$ h00f      $$ extra 0 required
```

This calculation yields the highest possible segment address and the lowest possible offset address.

The calculation to go from an existing segment:offset address to an absolute integer address is:

```
calc        segment   <= sysloc(global) $$ or local, as appropriate
            offset    <= varloc(variable)
            temp      <= ((segment $mask$ h0000ffff) $cls4$ 4)+
            (offset$mask$h0000ffff)
```

Accessing the Memory Pool

Data can also be transferred from variables to the TenCORE memory pool, as in:

```
memory write,mempool,moffset,bigbuf,bufsize
```

Where **write** is a keyword specifying the direction of the transfer; "mempool"is the name of the memory pool block; "moffset" is the offset into the memory pool block; "bigbuf' is the variable at which to start the transfer; and "bufsize" is the number of bytes to transfer.

The keyword **write** specifies that the transfer is from variable space to the memory pool; **read** transfers from the memory pool to variable space; and **exchange** swaps the contents of the two.

For more information on this topic or on primary and secondary memory pools, refer to the **memory** command.

Command List

area 171
asmcall 187
at/aff 189
beep 192
block 194
box 195
branch 197
calc 198
calcc 198
calcs 200
circle 201
clearu 202
clock 203
color 204
colore 205
compare 207
compute 209
date 210
debug 211
device 212
disable 213
do 216
dot 219
draw 220
ellipse 221
else 222
elseif 223
enable 223
endif 227
endloop 227
erase 228
error 229
exchang 230
exec 232

exitsys 236
extin 237
extout 239
fill 240
find 241
flow 244
font 261
goto 272
if 273
image 275
initial 284
intcall 287
loadu 296
loop 297
memory 299
mode 306
move 311
nextkey 312
nocheck 315
operate 315
origin 316
outloop 318
pack, packz packc, packzc 319
page 323
palette 325
pause 329
perm 336
polygon 341
press 342
print 344
put 346
receive 348
reloop 350
return 351
rotate 353
scale 355
screen 357
seed 363
set 364
setbit 365
setc 366
show 368
showa 370
showh 371
showt 372
showv 374
status 375
text 380

width....396
window....396
write....402
writec....404
zero....406

area

Manages pointer input areas.

area	keyword;...	
	define	defines a pointer area
	highlight	specifies highlight type and color
	disable	temporarily deactivates current pointer areas
	enable	reactivates disabled pointer areas
	clear	deletes current pointer areas
	select	highlights a pointer area
	save	saves current pointer areas to a named memory block
	restore	retrieves pointer areas from a named memory block
	delete	deletes named pointer area memory blocks
	reset	deletes all named pointer area memory blocks
	toggle	highlights or dehighlights a toggle pointer area
	inarea	determines if a location is within a pointer area
	dir	returns list of active pointer area identifiers
	key	returns the area identifier of a specified key

info	reports the attributes associated with a pointer area
repos	repositions all pointer areas in a window

Description

Facilitates pointer input within defined areas of the screen. Once a pointer area has been defined, a **Click** or other pointer action within the area generates a specific input value, just as if a keyboard key had been pressed. Appropriate coding can cause such an input to trigger a **flow** branch to another unit, break a **pause**, generate typing at an arrow or any other action desired.

Up to fifty active pointer areas may exist; this number can be increased with the **/sa=** command line option. The number of currently defined pointer areas is stored in the system variable **zareacnt**. **zmaxarea** holds the maximum number of areas allowed.

area definitions are cleared upon a **jump** branch to a new main unit. As part of the initializations for a new main unit, the default set of **areas** are activated. **area** definitions are saved when a window is opened and restored to their previous state when the window is closed unless **noarea** is used on the **window close**.

Pointer areas can be highlighted and dehighlighted automatically as the pointer moves in and out of the areas. The current pointer areas, together with their characteristics, can be saved in a named memory block; these can be deactivated and reactivated using the name. Pointer areas can be temporarily disabled and enabled.

area **define**; [*LOCATION*]; [*LOCATION*]; [*action=KEY, action=KEY...*]

Defines a rectangular screen area, whose opposite corners are identified by the two locations, within which any of the pointer actions listed generates an input key. Pointer areas with several defined actions and key values may be continued on following lines.

The following pointer actions can generate input values. Each pointer action has its own keyword:

ENSDOCID <WO 9907007A2 I >

click=	any pointer button is pressed
left=, right=	the left or right pointer button is pressed
clickup=	any pointer button is released
leftup=, rightup=	the left or right pointer button is released
add=, sub=	a second button is pressed or released
enter=, exit=	the pointer enters or leaves an area

If the keyword **left** or **right** is used, then **click** cannot be used. If **leftup** or **rightup** is used, then **clickup** cannot be used. If **click=** is the only action specified, the **click=** keyword can be omitted.

KEY may be any ordinary key value, such as a,b,c, %f1, etc. or one of the pseudo-keys %input1 - 999.

Example 1

Inputs the value **%enter** when a button is clicked within the area defined by 10:20; 11:30. In this case the keyword **click=** is optional (click=%enter). Coupled with the following **flow** command, the **area** input causes a **jump** to unit *nextunit*.

```
area     define; 10:20; 11:30; %enter
flow     jump; %enter; nextunit
```

Example 2

Inputs the value %"L" if the left button is pressed, and %"R" if the right button is pressed.

```
area     define; 100,200; 300,250; left=L,right=R
```

Example 3

Inputs the pseudo-key %input999 when the pointer enters the area defined by 5:3; 6:10. If the left button is released in the area, %f10 is generated.

```
area     define; 5:3; 6:10; leftup=%f10,enter=%input999
```

Example 4

Defines the entire screen as a pointer area. Clicking the right button will input the pseudo-key %other.

```
area     define; ; ; right=%other
```

Example 5

Defines a rectangular area from 10:10, 30 characters across for 2 lines. A pointer button click will input the %f1 key.

```
at       10:10
area     define; 30,2; ; click=%f1
```

Example 6

Defines a rectangular area around the text "Topic 1 : Introduction" allowing for fixed or variable spacing. A click while in the area will input %"1".

```
at       15:10
write    Topic 1: Introduction
area     define; 15:10; zx,zy+zfonth; click=%"1"
```

Example 7

Is continued over two lines.

```
area     define; 200,150; 400,170; left=a,right=%home,leftup=b, rightup=
«h01» ,enter=%input1,exit=%input255
```

area define; [LOCATION]; [LOCATION]; [action=KEY, action=KEY...] [;modifier; modifier ...]

Modifiers

The following modifiers can be used optionally with an **area define** statement. Any number of modifiers may be used and they can be listed in any order. **setid** and **getid** are mutually exclusive.

default preserves the active pointer areas for all subsequent main units. (Normally, a branch to a new main unit that erases the screen clears all pointer areas.) See also **Default Area Highlighting**.

priority assigns a priority from 0 to 15 in order to establish a precedence for overlapping areas. Level 15 is the highest priority.

If a priority is not set, the area is assigned a level of 0. If pointer activity occurs within two or more overlapping pointer areas, the highest priority area will take precedence. If overlapping pointer areas have the same level, the last defined area takes precedence.

getid specifies a variable to hold the unique 2-byte pointer area identifier that is automatically generated by the system when a pointer area is defined. System generated identifiers are always negative, between -32,768 and -1. The ID is used to identify a specific area in forms of the area command and is returned in system variable **zinputa**.

setid specifies a number which becomes the pointer area's unique identifier. Author-specified identifiers. must be between 1 and 32,767 to avoid a possible conflict with system-generated identifiers. The definition of a pointer area whose identifier is identical to that of an existing pointer area will result in the existing pointer area being redefined. The ID is used to identify a specific area in forms of the area command and is returned in the system variable **zinputa**.

Example 1

The pointer area definition will be a **default** area upon entering a new main unit. The area can be referenced later using the ID received in the variable *areaID*.

```
area     define; 5:10; 10:25; left=b,right=B; default; getid,areaID
```

Example 2

Inputs the pseudo-key %input1 when either button is pressed while in the rectangular area. The area can be referenced by the specified ID value 1.

```
area     define; 100,100; 200,110; %input1; setid,1
```

Example 3

Inputs the hexadecimal value 7000 when the pointer enters the rectangular area. If any other pointer area with a lower priority level overlaps this area, then this one takes precedence because it has the maximum priority level.

```
area     define; 400,150; 450,200; enter= «h7000» ; priority,15
```

area highlight: COLOR | off [; xor] [track | toggle]

Controls the highlighting color of all subsequent pointer areas. Area highlighting is off by default. A single **area highlight** statement affects all subsequent **area define** statements, but not pointer areas defined previously. Highlighting is reset to **off** by an **initial** statement.

The system variable **zareahl** holds the area id of the currently highlighted area (or 0 if no area is highlighted).

Highlighting to the given color is produced by plotting a box in an exclusive-or (XOR) mode and color over the screen area The upper-left pixel of the screen area is used in determining the box color that will XOR the screen to the desired highlight color.

Modifiers

The following modifiers can be used optionally with an **area highlight** statement but only when a color is specified. **track** and **toggle** are mutually exclusive.

xor specifies that pointer areas are to be highlighted by directly XORing the screen with the color specified

track dehighlights the area only when the pointer enters another area that has highlighting enabled

toggle disables automatic area highlighting and allows author-controlled highlighting (See **area toggle**.)

Default Area Highlighting

When default areas are in effect and a main unit is entered via a **jump** - type branch, area highlighting does not occur until after one of the following events:

- a branching command is executed (**loop, jump, jumpop, branch, doto, goto**)
- input is requested from the system (**pause, arrow, nextkey, delay**)
- the end-of-unit is reached
- certain commands that may have a long execution time are executed (**image, fill**)
- An **enable** command with the keyword **area, pointer** or **break**

These are the same points at which the pointer position is checked to see whether it has just entered or exited a highlighted area.

This delay allows the author time to get the display on the screen for which highlighting is desired. If a branch is required or if input must be requested before building the screen, the author must first issue a **disable area** statement to disable highlighting. After building the screen, **enable area** turns on any highlight.

Example 1

Produces red+ highlight areas. It uses the upper left corner of the area as an XOR reference color. Thus, the color of the pixel at 250,320 (the upper left corner of the area) is used to determine what color to XOR the area with to produce red+.

```
area     highlight; red+
area     define; 250,300; 400,320; click=%f10
```

BNSDOCID: <WO 9907007A2 >

Example 2

Turns off highlighting for all subsequently defined pointer areas.

```
area      highlight; off
```

Example 3

XORs subsequently defined pointer areas with blue. The areas will be highlighted when the pointer enters an area but will not be dehighlighted until it enters another area.

```
area      highlight; blue; xor; track
```

Example 4

Disables automatic area highlighting. **area toggle** can be used to highlight these areas.

```
area      highlight; white; toggle
```

area **disable [; arealD/s] [; noplot]**

Temporarily disables the pointer areas having the identifiers specified. The **disable** keyword alone temporarily disables all defined areas. A disabled pointer area no longer produces pointer input or a highlight. If a pointer area is highlighted when disabled, highlighting is turned off unless the **noplot** modifier is present.

Example 1

Disables the pointer area whose identifier is contained in the variable *idvar*. If the pointer area is currently highlighted, it will remain highlighted.

```
area      disable; idvar; noplot
```

Example 2

Disables the pointer areas whose identifiers are 1 and 2.

```
area      disable; 1,2
```

Example 3

Temporarily disables all pointer areas.

```
area      enable
```

area enable [; areaID/s]

Enables pointer areas previously disabled with **area disable**.

Example 1

Enables the pointer areas whose identifiers are contained in the array elements areaID(1), areaID(2), and areaID(3).

```
area      enable; areaID(1),areaID(2),areaID(3)
```

Example 2

Enables all pointer areas

```
area      enable
```

area clear [; [areaID/s][; noplot] [; default]]

When used with area identifiers, **clear** removes the specified pointer. When used without area identifiers, **clear** removes all areas. Areas are cleared regardless of being disabled.

If a pointer area is highlighted when it is cleared, the highlighting will be turned off unless **noplot** is used.

The **default** modifier is used to remove any area definitions that have been set with the **default** modifier.

Example 1

Deletes the pointer areas whose identifiers are contained in variables *ID1* and *ID2*

```
area      clear; ID1, ID2
```

Example 2

Deletes all pointer areas. A following **jump** branch will restore any unit **default** pointer areas.

```
area      clear
```

Example 3

Deletes all pointer areas from the entire screen, leaving any highlight on. When the window is closed, any previous areas are restored.

```
window   open; 100,100; 300,300
area     clear; ;noplot
...
window   close
```

area select; areaID | pointer | off

Highlights the area specified by the identifier or currently under the pointer. If **off** is specified, any highlighted area is turned off. **select** does not work with **toggle** highlight areas.

Only one area is highlighted at a time; any previously highlighted area is turned off.

Example 1

Highlights the area whose identifier is contained in the variable *aID*.

```
area     select; aID
```

Example 2

Turns off any highlighted area.

```
area     select; off
```

Example 3

Before opening the window, all areas are disabled. This turns off any highlight that is on. After closing the window, the areas are re-enabled and any area under the pointer is highlighted.

```
area     disable        $$ turn off areas and highlight
window   open, 100,.100;300,300
...
window   close
area     enable         $$ turn areas on again
area     select;pointer $$ highlight if pointer on area
```

area save; 'NAME' | local

area save; default

Saves the current set of area definitions in a memory pool block or the **default** buffer. The name can be either a text literal or contained in a variable. Named blocks can be restored later in any unit.

The **local** keyword saves the area settings in a memory pool block specific to the current unit. A local block can be restored only in the unit which saved it; it is deleted automatically when execution of the unit ends.

Saving the current area settings to the **default** buffer makes them the default **area** settings for all new main units. They are automatically reset on a **jump** to another unit.

The memory pool is used by the commands: **memory, image, window, status. area, flow, font** and **perm**. Memory blocks are tagged as belonging to a specific command type at creation and cannot be accessed by other commands using the memory pool; different commands can use the same name for a memory block without conflict.

Example 1

Saves the current pointer areas in the named memory block *level1*

```
area     save; 'level1'
```

Example 2

Saves the current pointer areas using the name contained in the variable *areavar*.

```
area     save; areavar
```

Example 3

Saves and restores the active flow settings over a subroutine call in a memory pool block unique to the executing unit. The block is automatically deleted when the unit is exited.

```
area     save; local
```

```
do        routines,graph
area      restore; local
```

Example 4

Makes the current area settings the defaults for all subsequent units that are entered by a jump branch.

```
area      save; default
```

area restore: 'NAME' | local [; delete] [; noplot]

area restore; default [; noplot]

Replaces the current area settings with a previously saved set. Optionally, the named or local block can be deleted from the memory pool by using the **delete** modifier.

Any areas highlighted when saved are highlighted when restored unless the **noplot** modifier is used.

Example 1

Saves and restores the current area settings over a library call. The library routine can alter the active area settings as desired without affecting the calling program upon return. Alternately, the **area save** and **restore** could be built into the library routine to provide a more easily used tool.

```
area      save; 'areas'
library   routines, graph
area      restore; 'areas'
```

Example 2

Saves and restores the active flow settings over a subroutine call in a memory pool block unique to the executing unit. The block is automatically deleted when the unit is exited. The highlight status is not restored.

```
area      save; local
```

```
do      routines,graph
area    restore; local; noplot
```

Example 3

Restores the unit default set of pointer areas.

```
area    restore; default
```

area delete; 'NAME'l local

Deletes a saved set of pointer areas from the memory pool without affecting any current pointer area definitions.

Example

Deletes from memory the areas saved under the name *menu.*

```
area    delete; 'menu'
```

area reset

Deletes all named sets of pointer areas from the memory pool without affecting current definitions. The **default** set of area settings are unaffected.

area toggle; arealD

Highlights or dehighlights the pointer area whose identifier is specified. This keyword option only operates on **toggle** pointer areas (as set by **area highlight**) It is intended for authors who want total control over the highlighting of pointer areas. Unlike **area select,** more than one pointer area can be highlighted.

Example

Highlights the pointer area whose identifier is contained in the variable *cID.*

```
area    toggle; cID
```

area inarea: LOCATION; arealD

Determines whether the specified location is within a pointer area. If so, the area's identifier is put into the variable *arealD*. If the location is within multiple areas, the highest priority area or, if levels are the same, the last defined area takes precedence.

If the location is not within any area, the variable is set to 0.

Example

Sets *whichID* to 1, the identifier belonging to the priority 15 pointer area.

```
area     define; 90,100; 120,160; click=%space;
         priority, 15; setid,1
area     define; 90,100; 120,160; click=%home;
         priority, 6; setid, 2
...
area     inarea; 100,155; whichlD
```

area dir; idBuffer [;length]

Returns the identifiers of all active pointer areas. To report on the maximum number of pointer areas, the area identifier buffer should be **zmaxarea***2 bytes in length.

Example

Returns the area identifiers of all currently active pointer areas in the array buffer *id*.

```
define   local
id(50),2          $$ use system default for array length
define   end
...
area     dir; id(1); 2*zareacnt
```

area key; KEY; arealD

Returns the identifier of the pointer area which includes the specified key in its pointer action list. If two areas have the specified key listed, then the identifier of the highest priority area

3NSDOCID: <WO__9907007A2 I >

is returned. If the areas have equal priority, then the identifier of the last defined pointer area is returned.

area info; arealD; infoBuffer48

Returns the attributes of an enabled pointer area into the buffer specified. The pointer area is selected by providing its identifier. The structure of the 48-byte buffer is as follows:

Attribute	Bytes
lower left x coordinate	2
lower left y coordinate	2
upper right x coordinate	2
upper right y coordinate	2
highlight color of area	4
type (0=none; 1=highlight; 2=track)	1
priority (0=none; 1..15=priority level)	1
default (0=none; 1=default)	1
window open level (**zwindows**)	1
reserved	16
left= key value	2
right= key value	2
enter= key value	2
exit= key value	2
leftup= key value	2
rightup= key value	2
add= key value	2
sub= key value	2

Example

Reads the information from the area whose identifier is contained in the variable *ID2*, into the buffer named *areadata*.

```
area     info; ID2; areadata
```

If a window is in effect when this statement is executed, the specified coordinates will be relative to the current window.

area repos; xOffset,yOffset

Adjusts all pointer areas by the absolute screen dimensions specified. If one or more windows are open, it operates only on areas created while the current window was open.

Example

Moves all pointer areas to the left 20 pixels and up 100 pixels.

```
area     repos; -20,100
```

System Variables

zreturn

The **save**, **restore**, **reset** and **delete** forms of the **area** command, which use the memory pool, report on success or failure in **zreturn.** In addition, the **select** and **toggle** forms also report error conditions through **zreturn.** All other forms set **zreturn** to ok (-1). The major **zreturn** values are:

-2	Redundant operation; area is already selected
-1	Operation successful
10	Name not found
11	Select or toggle of invalid highlight type
18	Unable to fill memory pool request

Miscellaneous

zareacnt	Number of currently defined pointer areas
zareahl	**area** id of currently highlighted area (else 0)
zinputa	**area** id of area causing the current **zinput** value (else 0), set when **zinput** updated
zmaxarea	Maximum number of areas that can be defined

asmcall

Calls an assembly language routine loaded into variables.

---

asmcall buffer [,variable]

buffer starting variable where routine is loaded

variable puts offset of specified variable into register BX

Description

Calls an assembly language routine which has been loaded into local or global variables. Segment register DS points to the start of variables. if the optional *variable* is used, register BX is set to the offset of *variable*. The result is that DS:[BX] points to the variable listed as the second tag. A far *return* must be executed at the end of the routine to return control to TenCORE. The value of the AL register is returned in the TenCORE system variable **zreturn**.

The assembly language routine may alter any of the microprocessor registers, but SS and SP, if changed, must be restored to original values before returning to TenCORE.

Addresses

The following system-defined function references are sometimes useful with **asmcall**:

sysloc(global) Segment address of global variables

sysloc(local) Segment address of the current unit's local variables

varloc(variable) Offset address of *variable*

absloc(variable) Absolute location of *variable* within memory, expressed as a 4-byte offset with no segment. This corresponds to the segment:offset address as follows:

segment = **absloc() ars** 4

offset = absloc() $mask$ h0f

If you have the segment and offset and wish to convert to the absolute location, use the following calculation:

absloc = ((segment **$mask$** h0000ffff) **$cls4$** 4) + (offset **$mask$** h0000ffff)

Example

Reverses the bits in a byte.

```
*
define   global
asmbuff(12),1          $$ buffer for assembly language program
asmdata  ,1       $$ integer byte to reverse
define   end
*
*        Load short assembly language routine by direct execution.
*
*        The routine begins with DS:BX pointing to the TenCORE
*        variable "asmdata".  The program loads and reverses
*        the byte, leaving it in register AL.  After the asmcall,
*        zreturn is set to the value left in AL.
*
*        The set command used here allows the programmer to
*        emulate the structure that would be used for assembler
*        programming.
*
set      asmbuff(1)     $$ mirror      proc   far
         h8a,     h27   $$ mov ah, [bx]       ; fetch byte
         h0b9, h08, h00 $$ mov cx,0008       ; shift 8 bits
*
```

```
*                $$ mirrori:
          h0d0, h0ec     $$ sh rah,01          ; move LSB to CF
          h0d0, hd0      $$ rcl al,01          ; shift CF into AL
          h0e2, h0fa     $$ loop          mirror1
*
          h0cb      $$ retf                      ; far return
*                $$ endp
*
calc      asmdata <= h55 $$ trial value to demonstate bit reversal

asmcall asmbuff(1), asmdata   $$ call routine, pointing to data
at        4:4
write     Original value = «h,asmdata» hex
          Reversed value = «h,zreturn» hex
```

System Variable

zreturn Holds the value of AL register upon return

at/atf

Sets the screen location and text margins.

---

at [LOCATION] [; LOCATION]

Description

Sets the current screen x-y location and the left text margin to the first *LOCATION* in the tag. The system variables zx, zy and **zmargin** are updated to this new location. If character coordinates are used to specify the **at** location, the **zx** and **zy** system variables contain the lower left graphic coordinates of the character cell.

A right text margin (which is initially set to the right screen or window border) can be set by specifying a second *LOCATION*. The right margin is found in the system variable **zrmargin**. Text that hits the right margin is automatically wrapped to the left margin. See **text margin** for the available text wrapping options. The right margin is ignored by the obsolete **charset** character plotting.)

The blank-tag form sets the left margin to the current screen location. This form is equivalent to the following statement:

```
at      zx,zy
```

The **atf** form of the command allows for the automatic adjustment of the **zy** position based on the height of the first following real plotting character. The top pixel of this first following real character is placed at the same y-position as the top of a standard font character. This action of **atf** provides for an apparent fixed top margin for text even if the text contains unknown starting embedded text attributes such as size and font. The adjustment to **zy** occurs in the following **write** or **show** statement when the first real character is plotted on the screen.

Examples

Example 1

Writes text starting at x = 10, y = 150.

```
at      10,150
write   This is line one of my text.
        This is line two at the same left margin.
```

Example 2

Writes text starting at line 7, character 3.

```
at      7:3
write   The line:character format is used for text more often than the
graphic x,y format.
```

Example 3

The **text margin** statement with **wordwrap** sets plotting to perform automatic wrapping of words at the right text margin stated in the following two location **at**. The second location on the **at** sets up a right margin at the 30th character position on the screen (or current window). The line number part of the second location is not used by the system plotter.

```
text      margin;wordwrap
at        5:10; 20:30
write     This text will appear on the screen as a paragraph only 20
characters wide regardless of how long the lines look in source code...
```

Example 4

By using **atf** commands, the top of the text from the two **write** statements appear to have a common top margin on the screen. The second **write** statement that has an initial embedded size 2 code sequence adjusts the system variable **zy** upon plotting the "S" so that the top of the "S" appears in the same location as the standard size font.

```
atf       10:10
write     Size 1 text
atf       10:30
write     Size 2 text
*         $$ Embedded size 2 at start of text
```

System Variables

The following system variables reflect the actions of the **at** statement:

zx	X-coordinate of screen location
zy	Y-coordinate of screen location
zline	Line number
zspace	Character number

zxycolor	Color of the pixel at screen location
zmargin	Left text margin
zrmargin	Right text margin

beep

Causes the speaker to generate a controlled beep.

beep	duration [, hertz]	
	duration	the length of beep, in seconds
	hertz	frequency in cycles per second

Description

Produces a tone of the frequency and length specified in the tag. If the frequency is not specified, it defaults to 1000 Hz.

Examples

Example 1

Produces a small, perhaps recognizable musical tune.

```
beep    .15, 261.3    $$ C
beep    .2, 349.66    $$ F
beep    .2, 440.88    $$ A
beep    .4, 523.25    $$ C
beep    .2, 440.88    $$ A
beep    .4, 523.25    $$ C
```

Example 2

Demonstrates user control of a **beep** command.

```
define   local
length   ,4,r          $$ length of beep
define   end
*
at       5:5
write    How many seconds do you want the speaker to beep?
arrow    6:5
store    length
.        at  8:7
.        write    Please enter a number, like 2,3 or 5
ok
.        beep     length
endarrow
```

Frequencies

A one-octave scale from middle-C has the following frequencies. Multiply or divide by two for other octaves:

C	261.63
C#	277.18
D	293.66
D#	311.13
E	329.63
F	349.23
F#	369.99
G	392.00
G#	415.30
A	440.00

A#	466.16
B	493.88
C	523.25

To calculate more precise frequencies, use a multiple of 110 as a standard "A" and multiply by the twelfth root of 2 for each half-tone increment.

block

block [LOCATION] ;[LOCATION] ;[LOCATION] [;[from page][,to page]]

Description

Copies a rectangular screen area from one location to another including across display pages. Operational only on 16- and 256-color screens. The block command works solely with absolute plotting coordinates; it is not affected by origin, rotate, or scale.

The first two locations specify the corners of the source area to be copied; if not specified they default to the entire screen. The third location is the upper-left corner of the destination; it defaults to the same position as the source area.

The fourth argument (frompage) is the source page number; it defaults to the current read page (zrpage).

The last argument (topage) is the destination page number; it defaults to the current write page (zwpage)

```
block    x1,y1; x2,y2; x3,y3   $$ copy one area of screen to another
block    x1,y1; x2,y2;; 2,1    $$ copy from page 2 to page 1
block    x1,y1; x2,y2;; 2      $$ copy from page 2 to current page
at       10:10
block    6,5;;5:10             $$ copy 6 chars,5 lines from 10:10 to 5:10
```

BNSDOCID: <WO__9907007A2_I_>

box

Draws a solid box or a frame on the screen.

box [*LOCATION*] [; [*LOCATION*] [; Xframe [, Yframe]]]

Description

Draws a solid box or frame on the screen in the foreground color. See the Command Syntax Conventions for a description *of LOCATION.* The **erase** command is identical to **box** except that it uses the background color for drawing.

The presence of a frame width argument signifies that a frame rather than solid box be drawn. The widths of the right and left sides are specified in pixels by the first frame argument while the second frame argument specifies the top and bottom widths. If the second frame argument is absent, the top and bottom frame width is set to best match the side width in aspect appearance on the screen.

Examples

Example 1

The entire screen (or window) is made blue with a white frame 5 pixels wide.

```
color    blue
box
color    white+
box      ;;5
```

Example 2

A solid rectangle 200 pixels wide by 100 pixels high is put on the screen in the foreground color. Graphic coordinates are used to specify each x,y coordinate for a location.

```
box      101,101; 300,200
```

BNSDOCID: <WO 9907007A2 I >

Example 3

The text in the **write** statement is given a red frame 2 pixels wide. The **text measure** command is used to determine the screen area that a following paragraph of text plots over on the screen. This information is returned in the system variables **zplotxl, zplotyl** for lower left corner and **zplotxh, zplotyh** for upper right corner.

```
text     measure        $$ initialize text measuring
  *
at       100,150
write    This is a paragraph of text of arbitrary
         length that will be nicely framed...
*
color    red+
box      zplotxl-2,zplotyl-2;zplotxh+2, zplotyh+2; 2
         $$ frame the text
```

Example 4

A blue box 20 characters wide and 10 lines deep is put on the screen starting on the 10th line. Since the **box** command does not change the system variables **zx** and **zy** as set by the **at** command, the text will also occur at the start of line 10.

```
at       10:1
color    blue
box      20,10
color    white+
write    This text appears in the box...
```

branch

Transfers execution to a labeled point in the same unit.

---

branch label | **q**

branch *SELECTOR;* label | **q**; label | **q**;...

label	Any valid label present in the current unit. A label is a numeral optionally followed by 1 to 7 letters or numerals, in the command field of a line.
q	Branch to the end of the current unit.

Description

Alters the flow of execution within a unit, by transferring execution to the line following the specified label, or to the end of the unit if the keyword **q** is specified. A blank entry in the selective list is used to fall through to the following command.

Example

If *name* is already set, the **branch** command will skip the arrow structure.

```
branch   name=0;;lskip
at       10:10
write    Enter your name.
long     8
arrow    11:10
storea   name
ok
endarrow
*
lskip
```

```
at        15:10
write     Welcome, «showa,name».
```

calc

Assigns a value or calculation to a variable.

calc	variable ⇐ expression

Description

The expression to the right of the assignment arrow can be any combination of constants, literals, variables, operators and functions so long as it evaluates to a single numeric value.

calc cannot work with variables greater than 8 bytes in size.

The command name does not need to be repeated for additional lines of calculations.

For more information on calculations, assignments, and variables, refer to the section **Variables, Calculations, and Data Transfer** in this document.

Example

Increment a counter, compute a percentage, and evaluate a complex expression.

```
calc      count     ⇐     count + 1
score     ⇐ correct / total
dist      ⇐ (1/2) * g * (t**2)
```

calcc

Selectively performs an entire calculation and assignment.

calcc	*SELECTOR;* var1 ⇐ exp1; var2 ⇐exp2;;... --

Description

calcc is a selective form of **calc** that performs one of a list of calculations, based on the value of the selector. Each calculation in the list can assign a different value to a different variable. if no calculation is to be performed for a particular value of the selector, a null tag (;;)is used.

The **calcc** statement can be continued in the tag field of subsequent lines.

Examples

Example 1

review is set if the selector *right* is 1; *score* is assigned if *right* is 2 or greater. Nothing occurs for a zero or negative value of *right.*

```
calcc    right;;; review ⇐ -1; score ⇐ (right/5)*100
```

Example 2

The value 4 is assigned to *section* if the selector *complete* is true; if false, no assignment occurs.

```
calcc    complete; section ⇐ 4;;
```

Example 3

column is assigned if *select* is 1, *apples* is assigned if *select* is 2, and *total* is assigned *50* if *select* is 3. Nothing occurs if *select* is negative, zero or greater than 3.

```
calcc    select,,,
         column ⇐ 22 + pointer;
         apples ⇐ bushels*47
         total  ⇐ 50;;
```

calcs

Selectively assigns a value to a variable.

calcs	*SELECTOR;* variable ⇐ exp1; exp2;;...

Description

calcs is a selective form of **calc** that evaluates one of a list of expressions and assigns it to a given variable. Based on the value of the selector, an expression is selected and evaluated from the list on the right of the assignment arrow and assigned to the specified variable. If no assignment is to be performed for a particular value of the selector, a null tag (;;) is used; the variable remains unchanged.

The **calcs** statement can be continued in the tag field of subsequent lines.

Examples

Example 1

The selector *month* is used to set the variable *days* to its proper value. The selective assignment is continued over two lines.

```
calcs    month; days ⇐;;31;28;31;30;31;30;
         31;31;30;31;30;31
```

Example 2

If complete is true, *section* is assigned the value 4; otherwise it is set to the evaluation of the expression *last*2*

```
calcs    complete; section ⇐ 4;last*2
```

circle

Draws a circle.

---

circle radius [,**fill** | angle ,angle2]

Description

Draws a circle of the specified radius with its center at the current screen location in the current foreground color.

The optional **fill** keyword produces a solid circle filled with the current foreground color.

The current screen location is not altered after a one-tag **circle,** so concentric circles can be drawn without resetting the screen location.

The three-tag form draws an arc. Arcs are always drawn counter-clockwise from *angle1* to *angle2*. As a result, different arcs are drawn depending on which angle is specified first. For example, **circle 100,0,90** draws a quarter of a circle, from 0 to 90 degrees. The statement **circle 100,90,0** draws three-quarters of a circle, from 90 degrees around to 360 (or 0) degrees.

For an arc, the current screen location is updated to the ending (last) arc position.

Example

Draws a circle centered on the screen; two arcs with the same center but different radii; then a filled circle.

```
screen   vga,graphics,medium,color
at       320,240        $$ set the center of the circle
circle   80             $$ draw a circle with radius 80
circle   60,0,90        $$ draw a quarter of a circle
at       320,240        $$ reset the center of the circle
circle   70,90,0        $$ draw three-quarters of a circle
```

```
at        320,240        $$ reset the center of the circle
circle    40,fill        $$ draw a filled circle in the center
```

clearu

Removes unit(s) from the unit cache.

---

clearu *[UNIT]*

clearu *SELECTOR; UNIT; UNIT;...*

Description

Units are automatically loaded into a virtual memory central unit cache as they are needed, and deleted as necessary to make room for new units. **clearu** can be used to manually clear specific units from the unit cache.

The blank-tag form removes all but the current unit from the unit cache.

clearu clears only units loaded from a file of the type corresponding to the current mode of operation. See **operate** for a discussion of modes of operation.

clearu is affected by the **edisk** command. Because multiple copies of a single file may be present on different drives, there may be multiple copies of a single unit in the unit cache. If **zedisk** = -1, units are fetched by searching all drives. All units matching the lesson/unit specification in the tag of **clearu,** and matching the current mode of operation (source or binary) are cleared from the unit cache. if **zedisk** ≠ -1, then only a unit from the current execution disk is cleared from the unit cache.

If no unit is cleared from the unit cache because no unit matched the unit reference and mode of operation specifications, **zreturn** is set to -2. If an attempt is made to explicitly clear the currently executing unit, **zreturn** is set to 19.

clearu is primarily used by utility programs to control which version of a unit is in memory or to control which units are deleted to make room for others.

System Variables

zreturn

-2	Redundant operation; no action taken
-1	Operation successful
19	Operation invalid, tried to clear the current unit

clock

Reads or writes system clock.

---

clock time4 [,**set**]

Description

Copies the information from the system clock into the specified 4-byte buffer. The **set** keyword sets the system clock to the time in the buffer.

The system clock is stored as 4 bytes; one byte each for the hour, minute, second, and hundredths of a second.

Example

Reads the system clock and displays the current time in hours and minutes.

```
define   local
sysclock,   4                $$ variables to hold time
.        hours,1
.        minutes,1
.        seconds,1
.        hundrths,1
```

```
define   end
*
clock    sysclock       $$ read the current time
at       5:5
write    The current time is: «t, hours,2»:«t, minutes,2»)
```

color

Sets the foreground color.

color	*COLOR*
color	*SELECTOR; COLOR; COLOR;...*
write	«**color** | **c**, *COLOR*»

Description

Selects the foreground color for subsequent text and graphics.

The + at the end of a keyword indicates the bright version of the color. The keyword **backgnd** gives the background color (last specified with the **colore** command). A numeric expression evaluating to a hardware color number may be used in place of a keyword.

Keywords set the first 16 colors (see the Command Syntax Description for keyword Colors). A color number must be used for colors coded higher than 15.

The default foreground color is **white**. The default can be changed using **color** followed by **status save; default.**

The color of text can also be changed by embedding text attributes into the tag of a **write** command.

CGA Restrictions

When using any of the CGA graphics displays on a composite color monitor, turning the thickness option on **(thick on)** produces truer colors.

screen cga, graphics, medium, color

screen cga, graphics, medium, mono

When using **screen cga,graphics,medium,** four separate groups of colors are available:

1	2	3	4
backgnd	backgnd	backgnd	backgnd
green	green+	cyan	cyan+
red	red+	magenta	magenta+
brown	brown+	white	white+

All colors on the display must be from one color group. If a color from one group is selected while colors from another group are on the display, all colors on the screen change to the corresponding color in the newly selected group.

Example

Displays yellow text in a blue box.

```
screen    vga
color     blue
box       5:10;10:50
color     yellow
at        6:15;10:50
write     A paragraph of yellow text...
```

System Variables

zcolor Color number if set by a color keyword name; -1 if set by a color number

zhcolor Hardware color number

SNSDOCID: <WO 9907007A2 I >

colore

Sets the erase color.

---

colore *COLOR*

colore *SELECTOR; COLOR; COLOR;...*

write «**colore** | **ce**, *COLOR*»

Description

Sets the erase color to be used for erasing, including: mode erase plotting, full screen erases, the **erase** command, and character background plotting in **mode rewrite**. To fill the screen or window with the erase color, a full-screen erase must be performed after execution of **colore.**

The color keywords set the first 16 colors (See the Command Syntax Conventions). A color number must be used for colors coded higher than 15.

A numeric expression evaluating to a hardware color number may be used in place of a keyword.

On CGA graphics displays, **colore** is subject to the same color restrictions as **color**. See the **color** command description for details. On CGA text displays, bright colors (8-15) are not available for **colore.**

The default erase color is **black.** This default can be changed using **colore** followed by **status save; default.**

Example

Erases the screen to blue after [ENTER] is pressed.

```
screen   vga
color    white+
```

```
at        5:5
write     Press Enter to see the background color change.
colore    blue
pause     pass= %enter
erase
at        5:5
write     The screen has been erased to change the
          background color.
```

System Variables

zcolore	Color number if set by a color keyword; -1 if set by a color number
zhcolore	Hardware erase color number

compare

Compares two text strings.

---

compare string1, string2, length, result

---

string1 the reference string, typically a variable or an up to 8-byte left-justified literal.

string2 the compare string, must be in a variable.

length number of bytes to compare

result-1 = identical; 0 = no match; >0 = last matching character.

Description

Compares two strings byte-for-byte. Variable boundaries can be crossed. The value returned into *result* indicates the success of the **compare**. If the strings partially match, *result* is set to the position of the last matching character.

In addition to *result*, **compare** sets **zreturn** to -1 if the strings match. If the strings do not match, **zreturn** is set to indicate which string is "alphabetically" less than the other: 0 if the first is less, 1 if the second is less. The ASCII codes for the first characters that do not match will be compared to determine which string is "numerically" less. For instance, if "aaa" and "aab" are compared, **zreturn** will be set to 0 because the third character "a" of the first string is less than the third character "b" of the second string.

System variables can not be used for either string.

Example

Checks a "password" before letting the user continue

```
define   local
input    ,8              $$ the user's entry
password = 'saturday'          $$ today's password
return   ,2              $$ value returned by compare
define   end
*
at       3:2
write    Enter the password of the day...
long     8
arrow    4:2
zero     input
storea   input
compare  password, input,8,return
ok       return
.        write          Good. Let's continue
no
.        write          Sorry, wrong password.  Erase and try again.
endarrow
```

System Variables

zreturn

-1	First string = second string
0	First string < second string
1	Second string > first string

compute

Computes the numeric value of a string of characters.

---

compute string, length, result

string	text variable containing the characters to numerically evaluate
length	the number of characters to evaluate
result	numeric variable to hold the result of the compute

Description

Converts a string into a numeric value. The string may contain leading or trailing blanks, a leading + or - sign, and a decimal point. The string cannot contain any other operators. If the conversion is successful, the value is stored in *result. If* **compute** fails (**zreturn** =0), *result* remains unchanged.

Example

Converts a string to a numeric value, performs an operation on the value, then converts it back to a string:

```
define   local
string,  10          $$ holds the string to convert
length   ,2          $$ length of the string
number   ,4,r        $$ real number for result
```

```
define   end
*
pack     string; length ;23.65
compute  string, length, number          $$ convert to numeric
calc     number <= number * 4.83
packz    string;;«show, number»          $$ convert to string
at       5:5
write    The new string is:«showa, string »
```

System Variables

zreturn

-1	Operation successful
0	No valid number was found in the string
1	Overflow error trying to store the converted number in the result variable. The result variable needs to be larger.

date

Reads or sets the current date.

---

date date4 [,set]

Description

Reads the date from the system clock and puts it in a variable. If the set keyword is used, the system date is set to the date stored in the variable.

The system date is stored as 4 bytes; 2 bytes for year, 1 byte for month, 1 byte for day.

Example

Displays the current date.

```
define   local
today,4
.        year, 2, integer
.        month, 1, integer
.        day, 1, integer
define   end
*
date     today
*
at       10:10
write    Today is $$$
writec   month;;;January ;February ;March ;April ;May ;June ;
         July ;AuguSt ;September ;October ;November ;December ;
write    «show,day», «show,year».
```

debug

Turns step-by-step program execution on/off.

debug	on | off
debug	*SELECTOR;* on | off ; on | off;...

Description

debug on stops execution of the current unit and enters the debugger. From the debugger, execution may continue either one command at a time (step mode) or until a specified command or unit is reached (breakpoint mode). **debug off** returns to normal execution.

The debugging tool is exited by:

- executing **debug off**

- pressing [ESC] to quit debugging
- exiting through the System Tools window
- returning to the editor (normally via [SHIFT][F2])
- an execution error

debug is normally functional only during source mode execution. It can be made functional in binary mode by choosing that option when making the binary. **debug** does not function with the student executor.

The debugger can also be entered while running a unit by pressing [SHIFT][2], then selecting **Debug** from the **System key options** menu.

device

Controls TenCORE device drivers.

device	device, function [, variable [,length]]	
	device	device number to call, in the range 0 to 255
	function	device-specific function number to call, in the range 0 to 255
	variable	data variable needed by the specified function
	length	length of *variable*, if other than the defined length

Description

device allows authors to control hardware devices not directly supported by the TenCORE language, but which are implemented through TenCORE device support files. These include special mouse, touch-panel, and interactive video features which go beyond the standard support built into TenCORE commands. The *variable* tag is used only if the function requires a data variable.

BNSDOCID: <WO 9907007A2 I >

Device Support Files

To use **device**, a TenCORE device support file is required. The device support file is executed(loaded) before starting the TenCORE executor, and remains resident in memory. Usage of **device** depends on the device support file being addressed.

System Variable

zreturn

Set to the processor register AL before returning. Some device support files use this feature to return status information.

disable

Disables system features (opposite of enable).

disable keyword/s	
pointer	disables pointer display and all pointer input
ptrup	disables pointer-up inputs
cursor	disables blinking cursor on text mode displays
arrow	disables plotting of the **arrow** prompt
absolute	disables absolute screen plotting
break	disables **break** modified **flow** branching from interrupting processing
area	disables **area** input
font	plots characters from charsets instead of fonts
mode	disables mode changes embedded in text

Multiple features can be disabled by a single disable statement. See the **Defaults** section for the initial state of these system features.

Description

disable pointer

Turns off display of the pointer and all input from the pointing device.

disable ptrup

Turns off button-up input from the pointing device.

disable cursor

Disables the blinking cursor for text mode screens.

disable arrow

Disables plotting of the **arrow** prompt character > and trailing space. User input occurs exactly at the location given in the **arrow** command.

disable absolute

Turns off the **enable absolute** plotting mode and causes subsequent screen graphics to be modified by **scale, rotate, window**, and **origin** commands.

disable break

Disables **break** modified **flow** branching from interrupting program execution. **flow** branching still occurs at the standard input processing locations (see the **flow** command for description). **break** is often disabled over a block of critical coding that must be completed in its entirety (e.g., file manipulations and their **zreturn** checks). Care must be taken to avoid disabling **break** while in a programming loop with no other means of exit. Since system routing keys such as [SHIFT][1] and [SHIFT][2] are turned off by **disable break**, a system hangup can occur.

disable area

Turns off **area** input. New **areas** can be defined while **area** input is disabled, but input from them will not occur until **area** input is enabled.

disable font

Plots characters from charsets rather than fonts. Charsets are an obsolete form of character definition.

disable mode

Turns off mode changes embedded in text.

Defaults

The system defaults for the **enable** and **disable** command features are set by the **initial** command to the following:

ON	area, arrow, break, font, mode
OFF	absolute, cursor, pointer, ptrup

Status of Disable Features

The current ON/OFF status of the system features affected by the **disable** command are found in the bitmap of the 4-byte system variable **zenable.**

zenable BIT	FEATURE
1	absolute
2	pointer
3	mode
4	cursor
5	ptrup
6	font
7	area
12	break
13	arrow

Use the **bit** function to examine **zenable**. A bit set implies ON while unset implies OFF.

For example:

bit(zenable,2) = -1 means **pointer** ON

bit(zenable,2) = 0 means **pointer** OFF

Example

All **flow** branching is turned off while attaching a file and checking for success.

```
flow     do; %f10; glossary; break        $$ a flow break branch
...
disable  break                            $$ prevent interruptions
attach   'afile'; nameset                 $$ attempt to open file
jump     zreturn;;erroru                  $$ check success
enable   break                            $$ re-enable flow breaks
```

do

Executes a unit as a subroutine.

do	*UNIT* [([argument/16s] [; return/16s])]
do	*SELECTOR; UNIT* [([argument/16s] [; return/16s])] ; ...

Description

Executes the specified unit as a subroutine. Upon completion of the subroutine, execution resumes with the command following the **do.** Subroutines can be nested up to 20 levels deep.

The unit which contains the **do** command is sometimes referred to as the *calling* unit; the subroutine is the *called* unit.

If a jump branch to a new main unit occurs while in a subroutine, execution does not return to the command following the **do** and the **do** stack is cleared.

do can pass up to 16 arguments to a subroutine and receive up to 16 arguments from a subroutine. A semicolon separates the send argument list from the receive argument list, while a comma separates arguments within a list.

The *called* unit must accept the passed arguments with a **receive** command (unless a **nocheck receive** command is in effect) or an execution error will occur. The *called* unit must return any required return arguments using the **return** command (unless a **nocheck return** is in effect) or an execution error will occur. See **receive, return,** and **no check.**

All arguments are passed and returned by value. However, the address of a variable can be sent as a value using the **varloc** ( ) function.

Examples

Example 1

When writing courseware, it is advantageous to put displays and routines that will be used in multiple places into one place that can be called as a subroutine when needed. Unit *header* can contain coding for a title bar that will appear on all pages for this segment of the lesson. Unit *diagram1* can plot an image and other graphics that will be common for this segment of the lesson. The **do** call to unit *navigate* in lesson *tools* is used to show the standard display of student navigation options available throughout all lessons in this series of lessons.

```
do      header                  $$ in current lesson
do      diagram1        $$ in current lesson
at      20:10
write   This diagram shows the system we will be studying
        next at the start of its combustion cycle.
do      tools,navigate          $$ in lesson tools
```

Example 2

The **do** statement passes the value 3 to the subroutine *month* to request that it return the numerically identified month's name and number of days. The do command's argument list is divided into two parts separated by a semicolon: the first part is for sending arguments, the second part is for receiving arguments.

```
define   local
name,8
days,1
define   end
do       month(3; name,days)        $$ get the name and days
*                                   in the 3rd month
write    «a,name» has «s,days» days.
*
unit     month
define   local
lnumber,1
lname,8
ldays,1
define   end
receive  lnumber                    $$ obtain month number argument
packzc   lnumber; lname ; ; ; ;January;February;Harch;April...
calcs    lnumber; ldays <= ;; 31; 28; 31; 30...
return   lname,ldays                $$ return the name and days
```

Example 3

The expression *score < 75* is used as a true/false selector to choose either unit *sad* or *smile* which graphically shows a student's status with a sad or smiling face image.

```
do       score < 75;sad; smile
```

dot

Plots a single pixel on the screen.

---

dot [LOCATION]

Description

Plots a single pixel on the screen in the current plotting mode and color, at the specified location. If no location is specified, the current location (**zx,zy**) is used.

The **dot** command is affected by **thick**. With **thick on**, all dots are plotted two pixels thick; with **thick off**, dots are plotted one pixel thick.

Examples

Example 1

Displays a dot at the location of the mouse pointer (**zinputx,zinputy**) when the pause is broken.

```
enable   pointer
pause    pass= %pointer
dot      zinputx,zinputy
```

Example 2

Draws a sine wave using **dot**.

```
define   local
x        ,8,r                $$ a real variable for the x-coordinate
define   end
*
loop     x <=    0, 2* π, 0.1
.        dot          50*x, 100+50*sin(x)
endloop
```

System Variables

zx	Current horizontal graphic screen location
zy	Current vertical graphic screen location

draw

Draws lines between screen locations.

---

draw [*LOCATION*]; [*LOCATION*]; [*LOCATION*];...

Description

Plots lines on the screen in the current plotting mode and color. A line is drawn between each location separated by a semicolon. When a double semicolon (;;) is reached, **draw** skips to the next location.

A single **draw** command can be continued on more than one line. If the tag starts with a semicolon, the line starts at the current screen location.

With **thick on**, all dots are plotted two pixels thick; with **thick off**, dots are plotted one pixel thick.

Examples

Example 1

Draws a box and a triangle on the screen with one **draw** command. Note the double semicolon to separate the two figures and that the drawing is continued over multiple lines.

```
draw     10,10; 80,10; 80,100; 10,100; 10,10;; 10,130;
         100,130; 10,190; 10,130
```

Example 2

Draws connected line segments at random locations on the screen (or window). The starting semicolon is used to start each draw at the current screen location which is the ending point of the previous draw.

```
loop
.       color          randi(1,15)        $$ choose random color
.       draw           ;randi(0,zdispx), randi (0, zdispy)
.       delay          .01
endloop
```

ellipse

Draws an ellipse.

---

ellipse *Xradius, Yradius* [, **fill** | angle1, angle2]

Description

Draws an ellipse (or portion of an ellipse) with the center at the current screen location. If the **fill** keyword is specified, the ellipse is filled with the current foreground color. The current screen location is not altered after a full ellipse is drawn: multiple ellipses can be drawn with the same center without resetting the screen location.

The two-tag form draws a full ellipse. The four-tag form draws only the arc of the ellipse specified by the starting and ending angles. Arcs are always drawn counter-clockwise from the first angle to the second angle. For example, **ellipse 100,60,0,90** draws one quarter of an ellipse. The command **ellipse 100,60,90,0** draws three-quarters of an ellipse. The current screen location is updated after an arc is drawn (a four-tag **ellipse**).

Example

Draws an ellipse and two arcs centered on the screen.

```
at        zxmax/2, zymax/2          $$ set the center of the ellipse
ellipse   95, 50                    $$ draw a full ellipse
ellipse   90, 40, 0, 90             $$ draw a quarter of an ellipse
at        zxmax/2, zymax/2          $$ reset the center of the ellipse
ellipse   70, 30, 90, 0             $$ three-quarters of an ellipse
```

else

Alternate case for if/else programming structure.

---

else

Description

Should the conditions of the previous **if** and **elseif** commands be false, the indented commands between **else** and **endif** are executed. An **else** command is not required in an if structure. For more information on if structures, see the **if** command description.

Example

```
if        score    > 90
.         write         Very Good!
elseif    score    > 75
.         write         OK, but you could do better.
else
.         write         You need more practice.
endif
```

elseif

Conditional case for if/else programming structure.

---

elseif *CONDITION*

Description

If the tags of the **if** command and all previous **elseif** commands evaluate false and the tag on the current **elseif** command evaluates true, the following indented commands are executed. Control then passes to the command following the **endif**. For more information on if structures, see the **if** command description.

Example

```
if      score   > 90
.       write           Very Good!
elseif  score   > 75
.       write           OK, but you could do better.
else
.       write           You need more practice.
endif
```

enable

Enables system features (opposite of disable).

---

enable keyword/s

enable *«vadable4»*

pointer	turns on pointer display and pointer input
ptrup	enables pointer-up inputs
cursor	turns on blinking cursor on text mode displays

arrow	re-enables plotting of the arrow prompt
absolute	turns on absolute screen plotting
break	re-enables **break** modified **flow** branching to interrupt processing
area	re-enables **area** input
font	re-enables **font** plotting
mode	re-enables mode changes embedded in text

Multiple features can be enabled by a single **enable** statement. See the **Defaults** section for the initial state of these system features.

Descriptions

enable pointer

Turns on display of the pointer and input from the pointing device.

enable ptrup

Enables button-up input. Unless **ptrup** is enabled, the pointing device generates input only when a button is depressed

enable cursor

Turns on a blinking cursor for text mode screens. This command has no effect on graphic screens.

enable arrow

Re-enables plotting of the **arrow** prompt character > and trailing space.

enable absolute

Turns on absolute screen plotting mode. Subsequent screen graphics are not modified by **scale, rotate, window** and **origin** commands. The **scale, rotate** and **origin** commands are effectively turned off while **window** becomes modified by **relative** off and **clip** off.

enable break

Re-enables **break** modified **flow** branching to interrupt program execution.

enable area

Re-enables **area** input. Any previously defined current **areas** including those defined while **area** input was disabled are now active.

enable font

Returns to using fonts instead of charsets for character plotting. Charsets are an obsolete form of character definition.

enable mode

Re-enables mode changes embedded in text.

enable «variable4»

The variable form of the **enable** command is used to enable/disable all features according to their bit settings in a 4-byte variable. The variable is usually a saved copy of the system variable **zenable** that is used to later reset all features back to a previous state. The feature bitmap is discussed in the following section **Status of Enable Features.**

Defaults

The system defaults for the **enable** and **disable** command features are set by the **initial** command to the following:

ON	**area, arrow, break, font, mode**
OFF	**absolute, cursor, pointer, ptrup**

Status of Enable Features

The current ON/OFF status of the system features affected by the **enable** command are found in the bitmap of the 4-byte system variable **zenable.**

zenable BIT	FEATURE

BNSDOCID: <WO 9907007A2 I

1	absolute
2	pointer
3	mode
4	cursor
5	ptrup
6	font
7	area
12	break
13	arrow

Use the **bit** function to examine **zenable**. A bit set implies ON while unset implies OFF. For example:

bit(zenable,2) = -1 means **pointer** ON

bit(zenable,2) = 0 means **pointer** OFF

Example

The absolute screen coordinates of the pointer are displayed at the pointer location. The coordinates and display are relative to the physical screen and not affected by any **window, origin, rotate** or **scale** commands.

```
enable   pointer,absolute            $$ turn on pointer and absolute
loop
.        pause                       $$ move mouse then press button
.        at      zinputx,zinputy     $$ at pointer location
.        write   <s,zinputx» - <s,zinputy»    $$ show coordinates
endloop
```

BNSDOCID: <WO 9907007A2 I >

endif

Ends if/else programming structure.

---

endif

Description

Marks the end of an if structure. Every **if** must have a matching **endif** at the same indent level. For more information on if structures, see the **if** command description.

Example

```
if      score   > 90
.       write           Very Good!
elseif  score   > 75
.       write           OK, but you could do better.
else
.       write           You need more practice.
endif
```

endloop

Ends loop programming structure.

---

endloop

Description

Marks the end of a loop structure. Every **loop** must have a matching **endloop** at the same indent level. For more information on **loop** structures, see the loop command description.

Example

Fifty *X's* are written on the screen.

BNSDOCID: <WO 9907007A2 I >

```
loop    index   <= 1,50
.       write   X
endloop
```

erase

Erases a box, frame or characters on the screen.

---

erase [*LOCATION*] [; [*LOCATION*] [; Xframe [, Yframe]]]

Description

Erases a box or frame on the screen to the background color. See the Command Syntax Conventions for a description *of LOCATION.* The **box** command is identical to **erase** except that it uses the foreground color for drawing.

The presence of a frame width argument signifies that a frame rather than solid box be erased. The widths of the right and left sides are specified in pixels by the first frame argument while the second frame argument specifies the top and bottom widths. If the second frame argument is absent, the top and bottom frame width is set to best match the side width in aspect appearance on the screen.

Examples

Example 1

The entire screen (or window) is erased to blue.

```
colore  blue
erase
```

Example 2

A rectangle 200 pixels wide by 100 pixels high is erased to the background color. Graphic coordinates are used to specify each x,y coordinate for a location.

```
erase    101,101; 300,200
```

Example 3

The text in a **write** statement is later erased after a pause. The **text measure** command is used to determine the screen area that a following paragraph of text plots over on the screen. This information is returned in the system variables **zplotxl, zplotyl** for lower left corner and **zplotxh, zplotyh** for upper right corner.

```
text     measure        $$ initialize text measuring
*
at       100,150
write    This is a paragraph of text of arbitrary
         length that will be erased...
*
pause
erase    zplotxl, zplotyl; zplotxh, zplotyh
```

Example 4

Ten characters are erased.

```
at       10:1
erase    10
```

error

Specifies an execution error unit.

error	[*UNIT*]
error	*SELECTOR* ; *UNIT* ; *UNIT* ;...

BNSDOCID: <WO 9907007A2 I >

Description

Sets the execution error lesson and unit. Whenever an execution error occurs, the system jumps to the unit specified with **error.** If no unit has been specified, a TenCORE system error message is displayed.

The blank-tag form clears the error lesson and unit. The setting of the error unit remains in effect until another **error** command is encountered.

The "error unit" is normally set by the router lesson when the system is started.

When **error** is executed, the current mode of operation, (binary or source), is saved along with the error lesson and unit names. If an error causes a branch to the error unit, the saved mode of operation is used to decide whether to fetch the error unit from a binary file or a source file.

System Variables

zerrorl	Name of the error lesson
zerroru	Name of the error unit

exchang

Exchanges the contents of two areas of memory.

---

exchang *keyword,* offset; *keyword,* offset; length

keyword **routvars | r**	Router variables
display | d	CGA screen display memory
global | g	Global variables
local | l	Local variables
sysvars | s	System data area
sysprog | p	System program area
absolute | a	Absolute memory location

offset	offset from memory base
length	number of bytes to exchange

Description

Exchanges bytes between two areas of memory. Up to 65,535 (64K-1) bytes can be exchanged. An offset of 0 specifies the beginning of the area of memory.

For relative addressing, TenCORE divides the computer memory into areas, each with its own keyword. A location is specified by providing a keyword, then an offset An offset of 0 specifies the first byte of that area.

Additional information about each keyword:

routvars	This is a 256-byte area of memory which can be accessed only via **exchang** and **transfr**. This area is used by the Student Router for storing data while a user is in an activity. if using this router, be careful not to modify the first 128 bytes of this area.
display	CGA screen display memory. The display memory for other display types (EGA, VGA, etc.) cannot be accessed with **exchang**.
sysvars	Do not use this area for data storage.
sysprog	Do not use this area for data storage.
absolute	Programs which reference absolute memory locations may not work in multi-tasking environments.

Because **exchang** can access any area of memory in the computer, it should be used with caution.

When specifying an offset in variables, the **varloc( )**function can be used to specify the offset of a particular variable.

BNSDOCID: <WO 9907007A2 I >

Similarly, an absolute memory location can be specified using the **absloc( )** function. Note, however, that **absloc()** may not work properly on multi-tasking operating systems such as Windows or OS/2.

Locally defined names have precedence over globally defined names when using **varloc( )**. This could cause unexpected results when **varloc( )** is used with **exchang.**

Examples

Example 1

The expression *varloc(scorvar)* specifies the beginning of global variables containing user scores.

```
exchang  global, varloc(scorvar); routvars,128; 128
*        Uses 2nd half of routvars
```

Example 2

Makes certain that the 2-byte value in *X0* is less than or equal to the 2-byte value in X1.

```
if       X0 > X1
.        exchang global,varloc (X0) ;global,varloc (X1);2
endif
```

exec

Executes a DOS program or command.

exec	keyword...	
	command	executes a program or command through the DOS command processor, or provides access to the DOS prompt
	file	executes a program directly

Descriptions

exec command [, all | memorySize]

Provides access to the DOS command line prompt. To return to TenCORE, type **exit** at the DOS prompt.

DOS memory not currently used by TenCORE is available for use by programs or commands entered at the DOS prompt. If the **all** option is specified, as much DOS memory as possible is made available by temporarily removing most of TenCORE. If a memory size is specified, the system attempts to free the specified number of kilobytes of memory. Note that the DOS command processor uses a part of this memory.

exec command [, all | memorySize]; buffer [,length]

The DOS command processor is loaded and passed the buffer to execute as if its contents had been typed at the DOS prompt. Features of the DOS prompt such as path search and batch file execution are available. If the length is not specified, the defined length of the buffer is used; unused bytes at the end of the buffer should be zero-filled.

DOS memory not currently used by TenCORE is available for use by the passed program or command. If the **all** option is specified, as much DOS memory as possible is made available by temporarily removing most of TenCORE. If a memory size is specified, the system attempts to free the specified number of kilobytes of memory. The DOS command processor uses a part of this memory.

exec file [,all | memorySize]; buffer [,length]

Executes *.EXE* and *.COM* files (only). The complete file name (including the *.EXE* or. *COM* extension), as well as drive and path must be specified. The features of the DOS command processor such as path search and batch file execution are not available. Any characters starting with the first space or slash *(/)* are passed as arguments to the executed file. if the length is not

BNSDOCID: <WO 9907007A2 I >

specified, the defined length of the buffer is used; unused bytes at the end of the buffer should be zero-filled.

DOS memory not currently used by TenCORE is available for use by the executed file. If the **all** option is specified, as much DOS memory as possible is made available by temporarily removing most of TenCORE. If a memory size is specified, the system attempts to free the specified number of kilobytes of memory.

The **file** form of **exec** is more efficient than the **command** form as the DOS command processor is not loaded. This form also allows access to error reports for the executed program (the **command** form provides error reports for the DOS command processor).

Examples

The following defines are used for all the following examples.

```
define   local
buf, 100
define   end
```

Example 1

Uses the **command** form of **exec** to allow any DOS directive or program name typed at an arrow to be executed. As much memory as possible is made available.

```
write    Enter DOS command to execute:
long     100
arrow
storea   buf            $$ store input into variable
ok
endarrow
exec     command,all;buf
writec   zreturn;;Error: zreturn =  «showt,zreturn»
```

Example 2

Uses the **command** form of **exec** to copy all TenCORE datasets in the current directory to drive a:. As much memory as possible is made available.

```
packz    buf;;xcopy *.dat a:
exec     command,all;buf
writec   zreturn;;Error: zreturn =  «showt,zreturn»
```

Example 3

Uses the **file** form of **exec** to call the Autodesk Animator animation program *QUICKFLI.EXE* in directory *D:\PROGS* to play the animation *BACKHOE.FLI* in directory *C:\ANIMS*. 250 kilobytes of memory are requested.

```
screen   mcga,medium
packz    buf;;d:\progs\quickfli.exe c:\anims\backhoe.fli
exec     file,250;buf
writec   zreturn;;Error: zreturn =  «showt,zreturn»
```

System Variables

The **exec** command sets the following system variables.

command form:

zreturn	reports on success of operation
-1	DOS command processor successfully executed.
0	insufficient memory to load DOS command processor, or unable to provide requested amount of memory.
1	DOS command processor (COMMAND.COM) not found.

file form:

zreturn	reports on success of operation
-1	program successfully executed

BNSDOCID: <WO 9907007A2 >

0 insufficient memory to load program, or unable to provide requested amount of memory.

1 program no found.

zerrlevl DOS ERRORLEVEL value as returned by executed **program**.

Both forms also set the system variables **zedoserr** and **zerrcode** which provide extended DOS error information.

exitsys

Exits to DOS or to the program which called TenCORE.

---

exitsys [error [,**noerase**]]

error 1-byte integer value passed to the DOS variable ERRORLEVEL

noerase the screen mode is not changed and the screen is not erased

---

Description

Returns control back to the program which called the TenCORE executor. Normally, the user is returned to the DOS prompt. If TenCORE was started via a DOS batch file, the next command in that batch file is executed. If *error* is specified, the value is passed to the DOS variable ERRORLEVEL, where it can be tested using DOS commands. If the **noerase** keyword is present, the screen mode is not changed and the screen is not erased. This keyword can only be specified when using the return value *error*.

ERRORLEVEL Values

TenCORE uses ERRORLEVEL values to indicate various error conditions. Authors are advised against using values 0 to 31 with **exitsys**.

0 Normal exit via exitsys (no tag)

1	Copy protection violation
2	Error in TenCORE command line options
3	Insufficient memory to execute TenCORE
4	Display driver not found
5	Error in display driver (e.g., invalid revision number)
6	Error in syntax of TenCORE environment string
7-15	Reserved for later use
16	Initial unit not found
17	No startup found in first unit executed
18	Initial unit version incompatible with executor
19	All drives excluded by TCDISKS or TCSEARCH
20-30	Reserved for later use
31	[SHIFT][7] pressed from generic Execution Error display

extin

Reads data from an external I/O port.

extin	port, buffer, length	
	port	address of the I/O port to read from
	buffer	buffer to receive data from the port
	length	number of bytes to receive

Description

Reads the specified number of bytes from the specified external I/O port. This command can be used to control external devices via a serial or parallel port. Consult the *IBM Technical*

Reference Manual for details on addressing and controlling the serial and parallel ports. **extout** is the counterpart to **extin**.

Examples

Example 1

Reads one byte of data from COM1.

```
define   local
rstatus  , 1            $$ the variable to read into
define   end
*
extin    1016,rstatus,1
*        read 1 byte from address 1016 (h3F8) into rstatus
```

Example 2

Uses **extin** and **extout** to send a byte to the **COM1** port using xon/xoff protocol. This routine could form the basis of a routine to control a printer.

```
define   local
status   ,1       $$ status flags for com1:
dr       ,1       $$ data ready, bit 0 of the status bits
*                 if this bit is set, there is data ready to read
rstatus  ,1       $$ a character read from com1:
tx       ,1       $$ transmitter status.  If 0, the port is
*                 ready to accept another character to transmit
xonoff   ,1       $$ xon/xoff flag.  If 1, xoff, else xon.
char     ,1       $$ a character to send out the port
*
lsr      = 1021         $$ line status register address
rxvar    = 1016         $$ receive variable register address
txvar    = 1016         $$ transmitter variable register address
```

```
define   end
*
loop
.        loop
.        .        extin       lsr,status,1
*        .        now check the first bit of the status,
*        .        see if xon/xoff was sent
.        .        calc        dr <= status $mask$ 1
.        outloop dr = 0       $$ no xon/xoff, so proceed
*        if there was xon/xoff, read it and set the flag
.        .        extin       rxvar,rstatus,1
.        .        calcs       rstatus=17; xonoff <= 0,1
.        endloop
reloop   xonoff = 1                  $$ xoff, so loop again
*        now check bit 5 of the status to see if it's ok to send
*        another character
.        calc           tx <= status $mask$ 32
outloop tx=32           $$ if ok, send it, otherwise loop again
endloop
extout   txvar,char,1          $$ send the character
```

extout

Writes data to an external I/O port

extout	port, buffer, length	
	port	address of the 1/0 port to write to
	buffer	buffer containing the data to send to the port
	length	number of bytes to send

BNSDOCID: <WO 9907007A2 I >

Description

Writes the specified number of bytes to the specified external 1/O port. This command can be used to control external devices via a serial or parallel port. Consult the *IBM Technical Reference Manual* for details on addressing and controlling the serial and parallel ports. **extin** is the counterpart to **extout.**

Examples

Example 1

Writes the contents of *rstatus* to COM1.

```
define   local
rstatus  ,1
define   end
*
extout   1016,rstatus,1 $$ writes rstatus to COM1 (h3F8)
```

Example 2

See the **extin** command description, Example 2 for an additional example.

fill

Fills any bounded area on the screen.

fill	[color [, boundary]]	
	color	color to fill with
	boundary	color which defines the boundary of the area to be filled

Description

Fills an enclosed area with a specified color starting at the current cursor position. If no *color* is specified, the area is filled with the current foreground color. If no *boundary* color is specified, the fill stops at any color different from the original color of the beginning point.

If the area to be filled is not closed the entire screen is filled.

The **fill** command usually follows an **at**, which sets the point at which the fill is to begin.

Fill does not use the current plotting mode: the specified color always appears regardless of mode.

Example

Produces a red circle inscribed with a bright white triangle outline. After the first delay, the top segment of the triangle is filled with cyan. After the second delay, the top segment becomes magenta. After the last delay, the entire triangle becomes solid blue with a white outline.

```
screen   vga
color    red
at       320,216
circle   65, fill
color    white+
draw     320,335; 200,150; 440,150; 320,335
delay    1
color    cyan
at       320,334
fill
delay    1
fill     magenta
delay    1
fill     blue, white+
```

find

Finds the position of a data object within a list.

find	object, length, buffer, entries, incr, return	
	object	data to search for
	length	length of the object in bytes
	buffer	variable to begin search
	entries	number of entries in list
	incr	length of each entry in buffer in bytes
	return	entry number of first match

Description

Searches a list of character or numeric entries for the desired object string. **find** will search for the object every *incr* bytes, from the beginning of *buffer* without attention to defined sizes or boundaries of variables. It considers *buffer* to be the first byte of a list whose characteristics are defined entirely by *entries* and *incr*.

If *incr* is negative, the search proceeds backwards from the end of the list to the start.

return indicates the position in the list where *object* was found, with the first entry being 1. If the object is not found, -1 is returned. The value in *return* is always relative to the beginning of the list, even if the search is backwards.

Examples

The examples below all assume the following defines.

```
.define   local
name,15          $$ name to search for
found,2          $$ position in list where "name" found
```

```
list(5),15          $$ list of names
define    end
```

In each example, *list* contains 5 names, each of which occupies 15 characters. The contents are as follows:

```
John miller....mark ho........sarah johnston.james heflin...lisa berger....
↑               ↑              ↑              ↑              ↑
1               16             31             46             61
```

Null characters have been shown as dots to improve their visibility, and numbers have been added to help in counting bytes.

Example 1... Literal Object

Searches for the name 'mark ho':

```
find      'mark ho',7,list(1),5,15,found    $$ found will be 2
```

Because the name has 8 or less characters, it can be supplied as the text literal 'mark ho'. Next comes the length, 7 characters. The start of the list is given as *list(1)*, and its length as 5 entries of 15 bytes each.

After the *search*, *found* contains the value 2 because 'mark ho' is found at the second position (not the second byte) in the list.

Example 2...Variable Object

Objects of more than 8 bytes cannot be given as text literals and must occur in variables. To locate 'james heflin':

```
packz     name;;james heflin
find      name,15,list(1),5,15,found
```

The variable *found* receives the value 4 because 'james heflin' is found in the fourth position.

Example 3...Byte-by-Byte

To find a last name.

```
packz     name;;miller
find      name,6,list(1),75,1,found
```

When this example is executed, *found* receives the value 6.

Note that the length of the name is given as 6 because this locates part of an entry, not an entire entry.

Similarly, the list is specified as 75 one-byte entries instead of 5 fifteen-byte entries. This causes **find** to look for 'miller' starting at every byte, not just at the start of each 15-byte entry. This illustrates that the object of the search can be longer than the nominal entry length provided for it.

Example 4...Backwards Search

To search backwards, a negative value is given for the entry length. This searches backwards from the end of the list for 'john', looking only at the beginning of each 15-byte entry.

```
find      'john',4,list(1),5,-15,found
```

Here, *found* receives the value 1. Although the search proceeded backwards from the end of the list, the position is always counted from the beginning of the list.

Example 5...Backwards Byte-by-Byte

Another example of backwards searching is the following.

```
find      'john',4,list(1),75,-1,found
```

In this case, an entry length of -1 is used, causing **find** to look at every character starting from the end of the list.

found receives the value 37, because the first 'john' found when searching backwards byte-by-byte occurs in 'sarah johnston'. If the search had been in the forward direction, 'john miller' would have been found first, and *found* would have received the value 1.

flow

Manages lesson flow by event-driven unit branching.

flow	keyword...	
	jump	defines event driven unit branch with screen erase
	jumpop	like **jump** but without screen erase
	do	defines event driven subroutine call
	library	like **do** but forces a binary subroutine call
	clear	clears active flow event settings
	save	saves active set of flow events
	restore	restores saved set of flow events to active status
	delete	deletes saved set of flow events
	reset	deletes all saved sets of flow events
	dir	returns list of active flow events
	info	provides flow branch data about active flow event

Description

Manages event-driven branching. Events can be keypresses, pointer inputs, time-ups, etc. A specific event can trigger a jump to a new main unit or a subroutine call.

Up to 50 different events can be active at a given time. A set of flow events and associated branches can be defined as the default environment for all new main units throughout a lesson. Advanced options can manipulate a lesson's flow settings so as to provide a clean connection to a router or library routines.

Flow events normally occur at waiting states within a lesson:

BNSDOCID: <WO 9907007A2 >

- at a **pause** that allows flow key input
- at an **arrow** that processes user keypresses
- at the completion of a unit's execution ("end-of-unit")

A sequence of coding will not be interrupted by a flow event at any other place unless the flow definition has been modified with **break** to allow for program interruption.

flow jump | jumpop | do | library; KEY/s; UNIT [modifier; modifier;...]

Defines a connection between a keypress (or other input event) and a branch to a unit. The first argument is a keyword specifying the type of branch. *KEY* is described fully in the Syntax Conventions section. Multiple keys separated by commas are allowed and when pressed cause a branch to *UNIT*. A total of 50 different events can be active as created by multiple **flow** commands. A given event is associated with one branch at a time. If the same event is defined in two different flow commands, the most recent definition takes precedence. Modifiers alter the standard operation of the command. Argument passing is not supported.

jump

Ends the current unit and branches to a new main unit. The following initializations are performed:

- the screen is erased to the default background color
- all **flow** events and **area** definitions are cleared and reset to their main unit defaults
- all display parameters are reset to their main unit defaults (as set by **status save; default**)
- the unit becomes the new main unit for processing and flow events

jumpop

Ends the current unit and branches without a screen erase ("op" stands for "on page") to a new main unit. Only the following initializations are performed:

- all **flow** events are cleared and reset to their new main unit defaults
- the unit becomes the new main unit for processing and **flow** events

do

Calls a unit as a subroutine. No initializations are performed. Return from the subroutine unit continues execution at the place the **flow do** event occurred. 20 levels of nested subroutine calls are allowed. **flow do** executes in source or binary mode depending on the mode of the calling lesson.

library

Calls a unit as a binary subroutine. No initializations are performed. Return from the subroutine unit continues execution at the place the **flow do** event occurred. 20 levels of nested subroutine calls are allowed. **flow library** is frequently used for calling third party software where the coding exists only in binary form.

Example 1

When [ENTER] is pressed, branches and passes control to *quiz1* in the current lesson. The screen is erased and all display settings, flow branches, and pointer areas are reset to their main unit defaults. When [F8] is pressed, *moreinfo* is called as a subroutine and control remains with the calling unit. Unit *moreinfo* would contain coding to add new information to that already on the screen.

```
write    Press Enter to Begin the Quiz.
         Press F8 for more information.
flow     jump; %enter; quiz1
flow     do; %f8; moreinfo
```

Example 2

Branches and passes control to unit *index* in lesson *aero1* when either [F1] or [ESC] is pressed. When [PG↑] is pressed, the binary subroutine *scroll* is executed. Control remains in the calling unit upon return from *scroll*

```
flow    jump; %f1,%esc; aero1,index
flow    library; %pgup; scroll
```

Example 3

Branches to the unit named by the variable *unitVar* when any of the keys a, b, c or the value contained in the variable *xkey* is input.

```
flow    jump; a,b,c, «xkey» ; «unitvar»
```

Example 4

Branches with a screen erase to *help* when either [F1] is pressed or a **Click** occurs on the associated area. Pressing [F2] or a **Click** on its associated area branches to *more* without a screen erase. In either case, control passes to the new unit.

```
area    define;  100,0; 150,20; click=%f1
area    define;  200,0; 250,20; click=%f2
flow    jump; %f1; help
flow    jumpop;  %f2; more
```

Generic Unit Names

Generic branch names are provided for common **jump** destinations to new main units:

=next	The next physical unit to the current main unit in the lesson. If none, the branch is ignored.
=back	The previous physical unit to the current main unit in the lesson. If none, the branch is ignored.
=first	The first executable unit in the current lesson.

BNSDOCID: <WO 9907007A2 I >

=last	The last executable unit in the current lesson.
=main	The current main lesson and unit as held in the system variables **zmainl,zmainu**.
=base	The main lesson and unit from which the last **base** modified **flow** branch occurred or as set by the **base** command. If none, the branch is ignored. The names of the current base lesson and unit are held in the system variables **zbasel,zbaseu**. Typically used for returning from a supplementary lesson sequence such as help to the main line of study. See **base** modifier on page $$$.
=editor	The Source Editor (with the current executing unit as the source block being editied) if executed by the authoring system; ignored by the student executor.
=exit	The lesson exit as set by the **exitles** command and contained in the system variables **zexitl,zexitu**. Student users are typically branched back to their routing system such as the TenCORE Activity Manager or DOS if none is present. An author testing a lesson is returned to the File Directory.
=system	Opens the system options window with the calculator, image capture, cursor, etc. [F2] is normally loaded with this branch as a TenCORE system default during authoring; ignored by the student executor.

Example 1

Branches to the unit following or preceding the current main unit in the lesson when [ESC] or [F6] is pressed. In the first or last unit of the lesson, the branch is ignored. This is an easy way to program the essential flow for a linear page-turning lesson.

BNSDOCID: <WO 9907007A2 I >

```
flow      jump; %enter; =next
flow      jump; %f6; =back
```

Example 2

When [ESC] is pressed, branches to the exit unit that was specified by the **exitles** command in a router or, if none, then DOS.

```
flow      jump; %escape; =exit
```

Unanticipated Input

A flow event can be defined for all unanticipated input at one of the natural waiting states by using the pseudo-key **%other.** This "other" event occurs only if the input cannot be applied to any other possible **flow** branch or input situation.

Example 1

Causes any key inputs that would otherwise be discarded by the system to branch to unit *record* which, say, records these unanticipated inputs for later study.

```
flow      do; %other; record; default; break
```

Example 2

At the **pause**, keys a, b and c continue execution of the unit. Any input specified in an active **flow** event such as [ENTER] and [F6] work as defined. Any other inputs branch to unit *continue.*

```
flow      jump; %enter; =next
flow      jump; %f6; =back
...
flow      jump; %other; continue
...
pause     flow=all;pass=a,b,c
```

Example 3

At the **arrow**, standard ASCII input such as a, b and c cause typing to appear on the screen as expected. Other **arrow** related functions such as erase, copy, judge-with-the-Enter key, etc. also perform as expected. Any defined **flow** event such as the branch to unit *aha* is active; unanticipated keys such as an undefined [F12] branch to unit *help*

```
flow      do; %other; help
flow      jump; %f3; aha
...
arrow     10:20
```

Modifiers

Modifiers on a **flow** statement alter the default action of the command. Multiple modifiers separated by semicolons can be used and in any order although some are mutually exclusive.

default Establishes the **flow** branch as a default setting for all subsequent new main units. **default** modified flow settings are cleared by an **initial** statement or by a **flow clear** statement with a **default** or **router** modifier. **default** settings for an entire lesson are often placed in the **+initial** control block of a lesson so that they are set with any entry to the lesson.

complete Delays putting the **flow** command into effect until all processing in the unit is completed including any **pause** or **arrow** statements. The **flow** works only at the end-of-unit. For example, users can be required to complete all **arrows** in a unit before branching to the next unit. The **complete** and **break** modifiers cannot be combined.

router Establishes the flow setting permanently over all following units. It is NOT cleared by an **initial** statement The **router** modifier allows management

BNSDOCID: <WO 9907007A2 I >

software (such as the TenCORE Activity Manager) to permanently set keys for routing use in an application. **router** flow settings can be cleared only by using **flow clear** with the **router** modifier

For example, a **router** modified exit branch using [SHIFT][1] is set by the TenCORE system at start-up of the student executor to return users to DOS. If the Activity Manager is used, it resets this **flow** branch to exit users back to a return unit in the Activity Manager before launching an application.

Great care should be taken in clearing **router** flow settings in an application lesson as the user can become stuck in the application. The **router** and **break** modifiers are best used together unless you are certain that your lessons accept the router exit key in all situations.

break Allows the **flow** branch to occur at any point, interrupting any programming or **pause** in process. A **break** modified **flow** branch will always work unless temporarily turned off by **disable break**. The main use for **break** is to interrupt a programming loop that does not check for keys. The **break** modifier is often used with the **router** modifier by a router to guarantee that the exit branch will always work in application lessons. **break** cannot be combined with **complete.**

When a **break** modified **flow do** or **flow library** event interrupts commands being processed (as in a programming loop), the current program statement is completed, the subroutine is executed, then processing continues with the next statement. When a break occurs at a timed **pause** or **delay** command, the timing is suspended until return from

the subroutine. During the subroutine call, the **flow** event is temporarily disabled to avoid recursion and re-enabled upon exit from the subroutine. In addition, the following system variables are saved and restored over the subroutine call: **zinput, zinputf, zinputx, zinputy, zinputa, zreturn.**

Great care must be taken to avoid unreliable results when using **break** with the **do** or **library** form of **flow.** Since a **break** interrupt can occur anywhere, variables (especially system variables) can be accidentally changed in the interrupt **do** or **library** call level making them invalid upon return from the call. The **break** attribute should be turned off over critical coding by use of a **disable break** statement. See the following Example 4.

clearkeys Upon branching, deletes all pending input from the input buffer. **clearkeys** eliminates keys that may have been "stacked-up" waiting for some event to end, such as a lengthy full-page display.

base The **base** modifier on a **flow** branch causes the main lesson and unit from which the branch occurs to become the base lesson and unit and stored in the system variables **zbasel** and **zbaseu**. When a **base** modified **flow** branch initiates a supplementary lesson sequence, such as help, the **=base** generic *UNIT* location can be used to return the user to the starting base location.

windowclose Closes the current window, if any, before executing the **flow** branch.

windowreset Closes all open windows, if any, before executing the **flow** branch.

operate Before branching, restores the execution mode in effect when the flow statement was encountered. (See the operate command.)

BNSDOCID: <WO 9907007A2 I >

binary	Before branching, changes to binary execution mode
source	Before branching, changes to source execution mode.
tpr	Before branching, changes to tpr (Producer) execution mode.

Example 1

Causes an immediate branch to the next linear unit in the lesson when the [ENTER] key is pressed. All subsequent main units will default to this flow setting. If this statement is to apply to the entire lesson, it should be placed in the **+initial** control block

```
flow     jump; %enter; =next; default; break
```

Example 2

Sets [ESC] to be a router exit key. It is kept permanently over **initial** commands and will **break** any programming situation (unless **break** is disabled), close all windows, and jump to *route, return.*

```
flow     jump; %escape; route,return; router; break; windowreset
```

Example 3

Branches to unit *ques6* if [ENTER] is pressed at an end-of-unit. Any **pause** or **arrow** must be completed first. Any other inputs "stacked-up" are removed.

```
flow     jump; %enter; ques6; complete; clearkeys
```

Example 4

A library unit *(clock* in lesson *routines)* is installed to show the user the time whenever [F10] is pressed or when a %timeup input occurs. Within the library routine itself, a **time 1** statement is executed before exiting so that the %timeup key will keep occurring and "refresh" the clock every second while the user remains in the main unit. Critical coding in the lesson that may be affected by unexpected interrupts should be protected by disabling the **break** attribute over the coding, then enabling it again after the coding.

```
flow     library; %f10,%timeup; routines,clock; break; default
...
disable  break                $$ protect critical coding
attach   'file';nameset       $$ perhaps unit clock also does attach
do       zreturn;;erroru
setname 'block'               $$ and setname
...
enable break                  $$ re-enable break
```

flow clear [; [KEYS/s] [; default | router]]

Deletes flow settings from the active list and optionally from the new main unit **default** and the **router** lists. The **default** modifier is used to remove any **flow** definitions that have been set with the **default** modifier. The **router** modifier is used to remove both **router** and **default** modified flow settings.

Example 1

Deletes all active flow settings except those modified with **router**. Any settings made with the **default** modifier will be re-activated upon a **jump** to a new main unit.

```
flow     clear
```

Example 2

Deletes any active flow settings for [ENTER] and [F6] unless they were defined with the **router** modifier. If either of these keys were defined with a **default** modifier, it will be re-activated upon a **jump** to a new main unit.

```
flow     clear; %enter,%f6
```

Example 3

Deletes [ENTER] from the active and **default** flow settings unless it has the **router** attribute

BNSDOCID: <WO 9907007A2 >

```
flow     clear; %enter; default
```

Example 4

Deletes all flow settings, except those modified with **router**, from both the active and default settings. An **initial** statement also performs this task.

```
flow     clear;;default
```

Example 5

Completely deletes [ENTER] as a flow setting regardless of how it was set.

```
flow     clear; %enter; router
```

Example 6

Deletes all flow settings regardless of how they were defined. Use with extreme caution for all the system **flow** settings such as [SHIFT][F1] are also removed preventing any exit unless otherwise provided for.

```
flow     clear;;router
```

flow save;'NAME' | local

flow save; default

Saves the set of active flow definitions in a memory pool block or the **default** buffer. The name can be either a text literal or contained in a variable. Named blocks can be restored later in any unit.

The **local** keyword saves the flow settings in a memory pool block specific to the current unit A local block can be restored only in the unit which saved it; it is deleted automatically when execution of the unit ends.

Saving the active **flow** settings to the **default** buffer makes them the default **flow** settings for all new main units. They are automatically reset on a **jump** or **jumpop** to another unit.

The memory pool is used by the commands: **memory, image, window, status, area, flow, font** and **perm**. Memory blocks are tagged as belonging to a specific command type at

creation and cannot be accessed by other commands using the memory pool; different commands can use the same name for a memory block without conflict.

Example 1

Saves the active flow settings under the name *flows* in the memory pool.

```
flow      save; 'flows'
```

Example 2

Saves and restores the active flow settings over a subroutine call in a block unique to the unit. The block is automatically deleted when the unit is exited.

```
flow      save; local
do        routines,graph
flow      restore; local
```

Example 3

Makes the active flow settings the defaults for all subsequent units that are entered by a **jump** or **jumpop** branch.

```
flow      save; default
```

flow restore; 'NAME' | local [; delete]

flow restore; default

Replaces the active flow settings with a previously saved set. Optionally, the named or **local** block can be deleted from the memory pool by using the **delete** modifier.

Example 1

Saves and restores the active flow settings over a library call. The library routine can alter the active flow settings as desired without affecting the calling program upon return. Alternately, the **flow save** and **restore** could be built into the library routine to provide a more easily used tool.

```
flow      save; 'flows'
```

BNSDOCID: <WO 9907007A2 I >

```
library  routines,graph
flow     restore; 'flows'
```

Example 2

Re-activates the flow settings whose name is contained in the variable *setUp*. The memory block is then deleted.

```
flow     restore; setUp; delete
```

flow delete; 'NAME' | local

Deletes a saved set of flow settings from the memory pool.

Example

Deletes the *quiz* set of saved **flow** events from the memory pool.

```
flow     delete; 'quiz'
```

flow reset

Deletes all named sets of flow settings from the memory pool. The **default** set of flow settings are unaffected.

flow dir; keyBuffer [,length]

Lists the keys for all the active **flow** events. The key inputs are returned as a sequence of 2-byte values (zero terminated if possible) into *keyBuffer*. These values can be used as input for **flow info** to obtain the full information about a **flow** event. The system variable **zflowcnt** holds the number of active **flow** events.

Example

Obtain the input key values for all the active flow settings. The inputs are returned as 2-byte values in a 50-element (maximum) array called *value*.

```
define   local
value(50),2
define   end
```

```
flow     dir: value(1), zflowcnt * 2      $$ read active flow keys
```

flow info; KEY; infoBufter [,length]

Returns the flow setting parameters for a specified key input value in the given buffer. The maximum length of information returned is 22 bytes. The optional *length* argument can limit the number of bytes returned. An extensive coding example using **flow info** is found in lesson **TCSAMPLE** unit **i/flow**.

parameter	byte(s)	values
branch type	1	0= not flow event 1= **jump** 2= **jumpop** 3= **do/library**
lesson	8	branch lesson name
unit	8	branch unit name
range	1	0= current main unit only 1 = **default** event for new main units 2 = **router** (permanent event)
active	1	0 = at normal wait states 1 = **complete** (at end-of-unit) 2 = **break** (interrupt processing)
execution mode	1	-2 = no change on branch -1 = to binary 0 = to source 1 = to tpr
window	1	0 = no change on branch 1 = window close 2 = window reset
base	1	-1 = set =base on branch 0 = no change

System Default Flow Settings

While running your lesson, [SHIFT][F1],[F2] and [SHIFT][F2] are pre-defined by the authoring system as:

```
flow     jump; %F1; =exit; break; router
```


BNSDOCID: <WO 9907007A2 I >

```
flow     jump; %F2; =editor; break; router
flow     jump; %f2; =system; break; router
```

Only [SHIFT][F1] is pre-defined by the student system as:

```
flow     jump; %F1; =exit; break; router
```

These flow settings may be redefined, or deleted using **flow clear** with the **router** modifier. However, be sure to provide an alternate means to exit your lessons; otherwise a user could become stuck with no exit even to DOS.

To temporarily save, clear and restore both the active and default flow settings over or within a **library** call, do the following:

```
flow     save;'currents'          $$ save active flow settings
flow     restore;default          $$ restore defaults to active
flow     save;'defsave'           $$ save default flow settings
flow     clear;;default           $$ delete all but router settings
...
library routines,stuff $$ all settings but router can be changed
...
flow     restore;'defsave'        $$ read back saved default settings
flow     save;default             $$ set them back as defaults
flow     restore;'currents'       $$ restore saved active settings
```

System Variables

zreturn

The **save, restore, reset** and **delete** forms of the **flow** command, which use the memory pool, report on success or failure in **zreturn**. All other forms set **zreturn** to **ok** (-1). The major zreturn values are:

- 1 Operation successful

10 Name not found

18	Unable to fill memory pool request

Miscellaneous

zflowcnt	Number of active flow settings
zmaxflow	Maximum number of flow settings allowed

font

Selects fonts for text display.

font	[*'LESSON',*] *'FONT'* [; fontNumber | **standard** [;**noload**]]
font	[*'LESSON',*] *'FONTGROUP' [;* fontNumber | **standard** [;**noload**]]
font	; fontNumber
font	**standard** [; **standard**]
font	**info**; infoBuffer [,length]

Description

A font is a named block in a lesson. It contains character designs corresponding to some or all of the ASCII character codes 0 to 255. A font group is also a named block in a lesson. It contains a list of related fonts with different text attributes.

The **text** command attributes of **size, bold, italic** and **narrow** interact with fonts and font groups. For a font, the appearance of the attributes is synthesized (except for **narrow**). When a font group is active, the system automatically selects the member font most appropriate for the text attributes enabled at any given time.

One font or font group is always designated as the **standard**. The standard font is the basis for character screen coordinates, even if some other font is currently selected for text display. For example, **at 5:10** specifies a screen position 5 standard character heights below the

BNSDOCID: <WO 9907007A2 I >

top, and 9 character widths from the left. A font group appropriate for the current screen resolution is automatically designated as the standard when an **initial** or **screen** command is executed. However, this standard may be overridden with a different font or font group, thus altering the character grid.

A font or font group can optionally be associated with a reference number. This number alone can later be used to reselect the font.

The **status** command saves and restores all **font** and **text** parameters: the selected font or font group, the **standard** font, the font number table, and text attributes. A **status save;default** statement is required to save these parameters over a **jump** branch to a unit.

Fonts and font groups are kept in the memory pool. There is no need to explicitly delete fonts from memory; when space is required for other memory pool objects, they are automatically deleted and reloaded from disk if needed again.

The system standard font groups exist in lesson **TCSTDFNT**. An additional set of decorative font groups is supplied in lesson **TCFONTS**. The Font Editor can be used to modify any of the fonts in these lessons, import fonts from other sources, or create custom fonts from scratch.

font ['LESSON',] 'FONT'

Selects a specific font from the current lesson or optionally from a named lesson. The font is brought from the disk to the memory pool and remains active until replaced by another **font** command or a **jump** branch to a unit which resets to the default font.

Example 1

The **initial** command causes all plotting parameters to be set to their system standard defaults including the **standard** font for the current screen resolution. Font *gothic* in the current lesson is then loaded and used for the following **write** text. Text attributes are then changed so

that the last **write** statement appears in a synthesized form of font *gothic*. The **zreturn** check indicates any system errors in loading the font from the disk. A **zreturn** check is recommended after any **font** command, but is omitted in the following examples to save space.

```
initial
write    This text uses the system standard font.
font     'gothic'
if       zreturn > -1
.        write  Error «s,zreturn» loading Gothic
endif
write     This writing uses character designs in font Gothic.
text     size; 2
text     bold; on
write     This writing uses a bold and double size font
         synthesized from font Gothic.
```

Example 2

Selects font *bur48* from the decorative font library in lesson **TCFONTS**. This is the 48 dot high version of burlesk.

```
font     'tcfonts','bur48'
write    This text is using Burlesk character definitions.
```

Example 3

Selects the font named in the variable *fontVar* from the lesson named in the variable *fontLib*

```
font     fontLib, fontVar
```

font ['LESSON',] 'FONTGROUP'

Activates a **font group** and selects a font from the group based upon current **text** attributes. Specific fonts can exist in a font group for all 72 combinations of the text attributes

ENSDOCID: <WO 9907007A2 I >

size (1 through 9), **italic, bold,** and **narrow.** The **text** command (along with its embedded and uncover code forms) sets these attributes and therefore determines which font is selected from the group. If a font with the required attributes is not found in the group, the closest match is used, and the nonmatching attributes are synthesized if possible. Attributes that can be synthesized are **size** (only 2 through 4), **italic**, and **bold**. The system variables **zfontret** and **zfontf** (see System Variables later) indicate how closely the current font selection from the group matches the text attributes currently in effect.

By looking at a **font** statement, it is not possible to tell whether a font or font group is referenced. However, the block's type is displayed on the Block Directory page.

Example 1

The font group *mine* has 4 fonts in it specifically designed for size 1 and 2 in both normal and italic but not bold. Text attribute uncover codes in the write statements select for italic and bold. The italic comes from one of the designed fonts in the group while the bold is synthesized.

```
font     'mine'
text     size; 1
write    This normal and italic text comes directly from
         specifically designed fonts but bold is synthesized.
text     size; 2
write    This size 2 normal and italic text comes directly from designed
         fonts while bold is synthesized starting from the size 2 normal
         and italic fonts.
```

Example 2

The size attribute causes the size 2 font to be chosen from each of the decorative font groups.

```
text     size; 2
```

```
font     'tcfonts','burlesk'
write    This writing uses the font group burlesk.
font     'tcfonts','opera'
write    This writing uses the font group opera.
```

font standard

Selects the standard font or font group. This is one of the system standard fonts found in lesson **TCSTDFNT**, unless the standard font has been overridden as described in the following section.

Example

```
font     'gothic'
write    This text is in Gothic
font     standard
write    while this is in standard.
```

font ['LESSON',] 'FONT' ; **standard**

font ['LESSON',] 'FONTGROUP' ; **standard**

The optional keyword **standard** on a font selection causes the named font or font group to become the new standard, overriding the system standard font group. Any subsequent **font standard** statements select this new standard font or font group until a **jump** branch.

To make the new **standard** font selection permanent for an entire lesson, use **status save;default** to make the selection part of the default display parameters that are restored upon a **jump** branch. Then, only a subsequent **initial, screen, status restore;standard**, or **font standard;standard** statement restores the system **standard** font group in lesson **TCSTDFNT**.

System variables **zcharw** and **zcharh** report the width and height of the standard font at size 1. These values are the basis for character screen coordinates.

Example

Variable *typeface* contains the name of the font to activate; it becomes the **standard** font. If the font is 32 pixels high and 32 pixels wide, then both the system variables **zcharh** and **zcharw** become 32. For **screen vga,medium** (480 pixels high and 640 pixels wide), the character coordinate grid now has 15 (480/32) lines and 20 (640/32) columns. The text for the **write** statement would appear at the graphic coordinates of x = 128 ((5-1)*32) and y = 416 (480-(2*32)). This character grid remains in force until the **standard** font is replaced.

```
screen   vga, medium
font     typeface; standard
at       2:5
write    Character position 5 on line 2.
```

font standard ; standard

Restores the system standard font group from lesson **TCSTDFNT.** This might be desirable if the standard font has been overridden earlier.

The lesson **TCSTDFNT** contains the system **standard** font groups and fonts, one named for each vertical screen resolution: the names begin with the letters **std** followed by the vertical resolution of the screen. For example, **std350** is for **ega,medium** screens, **std48O** for **vga,medium** screens, and **std6OO** for **vga,high** screens. Whenever a **font standard;standard** or **initial, screen,** or **status restore;standard** statement is executed, the current screen resolution is used to select the system **standard** font group. Since it is a font group, the font selected from the group depends upon the current text attributes.

The system standard fonts can be changed by editing the fonts and font groups in lesson **TCSTDFNT** and then making a new binary to replace **TCSTDFNT.BIN**.

font ['LESSON',] 'FONT' fontNumber [;noload]

font ['LESSON',] 'FONTGROUP' ; fontNumber [;noload]

Selects a font or font group and assigns it a unique number from 1 to 35. Numbered fonts can be accessed by an uncover code sequence embedded in text This allows switching between fonts within a single **write** command, for example.

In the Source Editor, entering the sequence [CTRL][A],[F],[1] - [9] enters the uncover codes for the first 9 numbered fonts while the sequence [CTRL][A],[F],[A] - [Z] enters the codes for the numbered fonts 10 through 35. In student mode at an **arrow**, a similar sequence of codes can be used for switching fonts but the default uncover code sequence starts differently: [CTRL][U],[M],[F],[A] - [z].

Numbered **font** statements can be included in the **+editor** control block of a lesson so that text with embedded font selection sequences appears in the correct fonts during editing.

Font number assignments are saved and restored by the **status** command.

A single font or font group may be assigned more than one number by using multiple **font** commands. If this is done, any of the numbers assigned to the font can be used to select it. However, a given number can be associated with only one font or font group at a time. The most recent assignment of a given font number overrides any earlier assignment.

The optional keyword **noload** is used to allow a font or fontgroup to have a font number association without actually loading the font into memory. This option can save disk read time at the start of a routine where multiple fonts need to be associated with font numbers. Later, the font will be automatically loaded as needed when referenced by its number embedded in **write** statements.

BNSDOCID: <WO 9907007A2 >

Example 1

Two numbered fonts are loaded. Uncover code sequences are used to switch between fonts within the text of the **write** statement. (Uncover code sequences become visible when display of hidden characters is turned on in the Source Editor.)

```
font    'gothic'; 2
font    'geneva'; 3
write   This text is Geneva, ■f3 switches to Gothic, then .■f2 back to
Geneva.
```

Example 2

In the **+initial** control block, the three numbered fonts are put into the default display **status** buffer. Any **jump** branch to a unit resets to these three numbered fonts so that they can be used through uncover codes in text of **write, pack**, etc. statements. If the **font** statements are also put into the **+editor** control block, any text with the font uncover codes (such as the **write** statement of this example) would be displayed in its chosen font during editing. The optional keyword **noload** is used to avoid any possible startup delays.

```
*                       in the +initial control block
font    'tcfonts','geneva'; 1       ; noload
font    'tcfonts','poster'; 2       ; noload
font    'mine','firework'; 3        ; noload
status  save; default
*                       in units throughout the lesson
write   ■f1 Geneva, ■f2 Poster, ■f3 Fireworks
```

font ;fontNumber

Selects a font or font group that was previously assigned a number.

Example

Two numbered fonts are loaded. Several **write** statements switch between using the two fonts by referencing their numbers alone.

```
font     'gothic'; 2
font     'geneva'; 3
write    This text is Geneva then
font     ;2
write     switches to Gothic then
font     ;3
write     back to Geneva.
```

font info; infoBuffer [,length)

Reads information about the selected font or font group into a buffer for the given length in bytes. If the length is not specified, the defined length of the buffer is used.

The following 256 bytes of information are available:

System Data (100 bytes)	Bytes	Offset
font lesson name	8	0
font block name	8	8
font group lesson name (font lesson if font loaded)	8	16
font group block name (font block if font loaded)	8	24
font number (-1 =standard, 0=unnumberd)	1	32
font group (-1=yes, 0=no)	1	33
character width	2	34
character height	2	36
baseline offset	2	38
underline offset	2	40
underline thickness	2	42
underline gap	2	44
shadow x offset	2	46
shadow y offset	2	48
italic angle (real)	4	50
spacing decrement (0=no, 1=yes)	1	54
(reserved)	45	55

BNSDOCID: <WO___9907007A2_I_>

Descriptive Information (156 bytes)	Bytes	Offset
font name	16	100
revision number	1	116
family	1	117
subclass	1	118
style	1	119
weight	2	120
points	2	122
horizontal resolution	2	124
vertical resolution	2	126
(reserved)	48	128
copyright	80	176

Example

The copyright message for the **standard** font is displayed.

```
define   local
infobuf,256
.        ,176                $$ skip other fields
.        copyrt,80           $$ copyright field
define   end
...
font     standard
font     info; infobuf
at       10:1
showa    copyrt,80
```

System Variables

zreturn

After a **font** command, **zreturn** indicates the success of loading a font:

-1	Operation successful
0	Disk error (see **zdoserr, zedoserr**)
2	No font in group

4	File not found
10	Font name not found
15	File directory error - unrecoverable
18	Unable to fill memory pool request

zfontret

The system variable **zfontret** indicates how closely the current font matches the text attributes currently in effect.

zfontret for a **font:**

-1	Font loaded successfully
2	Unable to load font; **standard** font used
3	Unable to load **standard** font; charsets used

zfontret for a **font group** (and updated when text attributes change):

-2	Exact match of attributes; some synthesis used
-1	Exact match of attributes
0	Partial match of attributes
1	No suitable match; base font used
2	Unable to load base font; **standard** font used
3	Unable to load **standard** font; charsets used

zfontf

For **font groups** only, **zfontf** contains a bit field that reports on how the attributes are produced whenever a font is loaded or text attributes change causing the selection of another font in the group.

zfonff bit	**meaning**
6	Bold matched in font group

7	Italic matched in font group
8	Size matched in font group
16	Narrow matched in font group
22	Bold synthesized
23	Italic synthesized
24	Size synthesized

The attribute may not match the request but also may not be synthesized; for example, if the font group only has bolded items, and the current attributes specify non-bolded, then bits 6 and 22 will both be off.

Miscellaneous

The following system variables are set after loading a font and are updated if appropriate when text attributes change:

zfonth	Character height and width of selected font
zfontw	
zcharh	Character height and width of **standard** size 1 font
zcharw	(basis for the character coordinate system)
zfontb	Baseline offset of selected font
zcharb	Baseline offset of **standard** size 1 font

goto

Transfers execution to another unit or to the end of unit.

goto	*UNIT* | **q**
goto	*SELECTOR*; UNIT | **q**; UNIT | **q** ; ...

Description

Allows the logical continuation of the current unit into another unit. **goto** switches execution to the specified unit but does not: erase the screen, clear the **do** return markers, change the main unit, or alter a help sequence.

The **q** keyword transfers execution to the end of the unit.

The **goto** command is considered by many as an obsolete programming method properly replaced by structured programming.

if

Begins an if structure.

---

if *condition*

Description

Marks the beginning of an **if** structure. If the condition is true, any following indented code is executed. If the condition is false, execution continues with the next non-indented **elseif, else**, or **endif.**

An **if** structure is the main conditional structure in the TenCORE language. The commands which make up an **if** structure are:

if

elseif (optional, any number)

else (optional)

endif

The **if** structure begins with an **if** and ends with an **endif.** Commands to be executed when a particular condition is true are indented immediately following the **if, elseif**, or **else**. Indenting in

BNSDOCID: <WO__9907007A2_I_>

TenCORE consists of a period in the first character of the command field followed by 7 spaces (one tab stop in the source code editor). The command and tag of the indented command are then typed as usual.

In any **if** structure, only one set of indented commands is executed. The condition is checked on each **if** and **elseif** in succession. The indented commands beneath the first **if** or **elseif** that is true are executed and then control passes to the command after the **endif.** if none of the conditions on **if** and **elseif** are true, the indented commands following any **else** are executed.

Examples

Example 1

If the value of the variable *score* is over 75, the indented commands are executed.

```
if      score > 75
.       do goodfb
.       jump    endless
endif
```

Example 2

elseif and **else** commands are used in an **if** structure.

```
if      right = 0           $$ user missed all questions
.       do      verybad
.       jump    helppg
elseif  right < 5           $$ user got 1 to 4 right
.       do      notgood
.       jump    quiz
elseif  right < 10          $$ user got 5 to 9 right
.       do      good
.       jump    review
else                        $$ user got more than 9 right
```

```
.       do      great
.       jump    endless
endif
```

Example 3

An if structure is nested.

```
if      wrong > 5                   $$ user missed more than 5
.       write           You are making too many mistakes!
.       pause
.       if      wrong > 10  $$ user missed more than 10
.       .       jump quit
.       endif
endif
```

image

Displays, captures and manages screen images.

image	keyword	
	plot	displays the specified image
	save	captures all or part of the screen to the specified destination
	move	transfers an image between storage areas
	info	returns information about an image
	delete	deletes the specified image from the memory pool
	reset	deletes all images from the memory pool
	compress	turns compression of image data on or off
	palette	reads palette information from an image

BNSDOCID: <WO 9907007A2 I >

Description

The **image** command is used to plot completed pictures onto the display, to capture pictures off the display or to move pictures between variables, memory pool blocks and disk storage locations (namesets, datasets, and lessons). Pictures are screen images that are: created in image blocks by the Image Editor imported into image blocks from existing bit-map images created by "paintbrush" programs or a part of a screen that is directly captured and saved by either the Image Option -after pressing System Key [F2] or executing an **image save** command.

image plot; from [;[LOCATION] [;palette]]

Plots the specified image at the same location it was saved from, unless a location is specified. In either case, the current screen location is not changed. The display mode as set by the **mode** command is used in plotting the image. (See **mode** for additional information.) if the keyword **palette** is specified, the palette stored with the image is used. If **palette** is not specified, the palette is not changed. Images plot downwards from the specified location, which corresponds to the upper left corner of the image.

from keywords and syntax:

block | b ,*'NAME'*

Plots an image block from the specified lesson. If only the image name is specified, the current lesson is assumed.

file | f [,record]

Plots from the attached dataset or nameset, starting at record 1 unless a starting *record* number is specified. In the case of a nameset, a valid name must have been previously selected with the **setname** command.

memory | m ,'NAME'

Plots from a memory pool block. The name may be a literal in single quotes, or occur in a variable.

vars | v ,variable [,length]

Plots from variables. The defined length of the variable is used unless a *length* is specified.

Examples

```
image    plot;block, 'horse';20,100
*        block "horse" at location 20,100

image    plot;block, 'library','horse'
*        block "horse" in file "library"

image    plot;file
*        currently attached file

image    plot;memory,myimage
*        name contained in the variable myimage from memory pool
image    plot; vars , picsave
*        image in variable "picsave"

image    plot;memory,'show';15:01;palette
*        plot using the palette stored with the image
```

image save; destination [; [LOCATION] ; [LOCATION][;palette [,variable]]]

Saves the area of the screen specified by the rectangular area to *destination.* If no display area is specified, the entire screen is saved. If the **palette** keyword is used, the current palette is stored with the image as well. If *variable* is given, only those colors specified in *variable* are

BNSDOCID: <WO___9907007A2_1_>

saved with the image. See **palette read** for the format of *variable*. The image is saved in a special compressed format in order to use as little storage as possible.

destination keywords and syntax:

file | **f** [,record]

Saves to the attached dataset or nameset, starting at record 1 unless a starting *record* number is specified. In the case of a nameset, a valid name must have been previously selected with the **setname** command. The number of records required to save an image can be calculated as: (bytes + 255*)* $idiv$ 256.

memory | **m** ,*'NAME*

Saves to a block in the memory pool. The name may be a literal in single quotes, or occur in a variable. If an image is already stored with the specified name it is replaced. Names in the memory pool created by **image** are independent of any names created with other commands. For instance, the **memory** command and the **image** command can both create a name in the memory pool called *cat* without causing a conflict. The **initial** command deletes all images from the memory pool.

vars | v ,variable [,length]

Saves to variables. The defined length of the variable is used unless a *length* is specified. The image may be truncated if the defined length of *variable* is insufficient

Examples

```
define   local
palblock(16),6
         palslot,2
define   end
```

```
image     save;file;50,20;150,80
*         save in attached file

image     save;memory,'pic';0,0;99,99
*save in "pic", memory pool

image     save;memory,'save';0,0;zxmax,zymax;palette,palblock(1),10
*         save the first ten entries of the palette
```

image move; from; destination

Copies an image between a *from* and a *destination* when neither is the screen.

Example

```
image     move;block,'abc';memory,'def'
*         from "abc' in current file to block "def" in memory pool
```

block cannot be used for *destination*, since that would modify a running lesson source file. Also, if *destination* is **file**, then *from* can only be **memory** or **vars**.

image info; buffer; from | last [; palette]

image info;bufter; display | d, LOCATION [;[LOCATION] [;palette]]

Returns information about an image specified by *from*. The information is returned as 24 bytes in the specified buffer in the following fixed format:

1 byte: image type

0 = byte-oriented (CGA, EVGA, etc.)

1 = plane-oriented (EGA, VGA, etc.)

2 = text image

1 byte: bits per pixel (byte-oriented) or number of planes

2 bytes: bias from left of screen

BNSDOCID: <WO__9907007A2_I_>

2 bytes: bias from top of screen

2 bytes: image width

2 bytes: image height

4 bytes: number of bytes used by image block

10 bytes reserved

Bias, width, and height are given in pixels for byte- and plane-oriented images, and in character positions for text images.

Adding the **palette** keyword causes palette information to be included in the returned length of the image.

Additional *from* keywords and syntax exits for **image info:**

display | **d,** *LOCATION* [; [*LOCATION*] [; **palette**]]

The image is taken directly from the screen. This form can be used to determine how much space a particular image on the screen will require, before creating a file to hold the image.

last [;palette]

The last image processed by the **image** command.

Example

```
define   local
infovar,24
.        type,1
.        bits,1
.        xstart,2
.        ystart,2
.        xsize,2
.        ysize,2
.        length,4
```

```
.          ,10
define     end
*
image      info;infovar;display,50,50;100,100
at         2:1
write      Image will require «s,length» bytes for storage
pause      pass= all
created    'image1',(length+255) $idiv$ 256
do         zreturn;;error('created')   $$ always check zreturn
image      save;file;50,50;100,100
```

image delete; 'NAME'

Deletes the specified image from the memory pool.

image reset

Deletes all images from the memory pool.

image compress; [on | off]

Toggle image compression. Compressed images occupy less memory or disk space; non-compressed images in memory plot faster. The default is **on**.

image palette; from ; buffer, entries

Reads image palette entries into a buffer. The buffer must contain one or more 6-byte palette entries in the form of:

slot	(2bytes)
red	(1 byte)
green	(1 byte)
blue	(1 byte)
intensity	(1 byte)

BNSDOCID: <WO__ 9907007A2_I_>

The value of *slot* must be pre-set for each palette entry before executing **image palette**. The value of each slot will determine which palette slot's information is read into that entry. For example, if the value of *slot* for a particular entry is 8, the information for palette slot 8 will be read into that entry. If the values of *slot* are not set, slot 0 will be read into all the entries. Any subset of palette entries may be read or written by setting appropriate slot values.

entries determines how many entries will be read.

Example

Reads the palette information for the base 16 palette entries from the image in memory block *test:*

```
define   local
pvar(16) ,6
.        slot,2
.        pred,1
.        pgreen,1
.        pblue,1
.        intensty,1
count    ,2
define   end
*
loop     count <= 1, 16
*must initialize slot numbers
.        calc     slot(count) <= count - 1
endloop
*
image    palette; memory, 'test';pvar(1),16
```

Screen Compatibility

To be able to display an image from one screen on another screen, they must be of the same type. There are three types of image:

- graphics byte-oriented, such as the CGA, Hercules, MCGA, and EVGA
- graphics plane-oriented, such as the EGA, and VGA
- text

For graphics screens, the number of bits per pixel (byte-oriented) or number of planes (plane-oriented) must also be identical. Text images are compatible among all text screen types.

The characteristics of a given image can be determined using the **image info** command. The following tables list the characteristics of the graphics screens currently supported by TenCORE:

Byte-oriented Screens	Bits/pixel
cga,medium	2
cga,high	1
hercules	1
mcga,high	1
mcga,medium	8
evga	8

Plane oriented Screens	Number Of Planes
ega	4
vga	4

For example, in the byte-oriented group, the cga high, hercules, and mcga high images are compatible with each other. In the plane-oriented group, all listed screens are compatible with each other. ***Caution:*** *even though an image from one screen is compatible with another screen, it may look different on the other screen due to changes in aspect ratio.* For example, an ega high image plotted on a vga medium screen will be the same width, but only be about 73% of the original height.

System Variable zreturn	
-1	Operation successful
0	Disk error (see zdoserr, zedoserr)
1	No file attached
2	Block out of range
3	Memory out of range
4	File not found
5	Image screen type doesn't match execution screen type
8	Insufficient disk records
9	No name in effect
10	Name or block not found
11	Invalid type
16	Invalid name
17	Invalid image
18	Unable to fill memory pool request

initial

Initializes the system to a standard state.

---

initial [nodetach]

Description

Sets the system to a standard state. It is normally used when starting a lesson, as in a **+initial** control block, to set all system parameters to a known state.

The **nodetach** keyword prevents the currently attached file from being detached, and preserves any locks which are in effect for shared files.

The **initial** command performs the following command initializations:

Display	
window	reset; noplot
status	restore; standard
blink	off
color	white
colore	black
colorg	black
disable	cursor
enable	font
font	standard; standard
mode	write
origin	0,0
rotate	0
scale	1,1
text	align; baseline
text	bold; off
status	save; default
palette	init
disable	absolute
text	delay; 0
text	direction; right

BNSDOCID: <WO ... 9907007A2 I >

	text	increment; 0,0
	text	italic; off
	text	margin; wrap
	text	narrow; off
	text	reveal; off
	text	rotate; 0
	text	shadow; off,white
	text	size; 1
	text	spacing; fixed
	text	underline; off,foregnd
	thick	off
	page	1
	display	1
	Input	
	disable	pointer
	disable	ptrup
	enable	area
	area	clear; default
	area	highlight; off
	uncover	%ctl"u"
	force	no lock
	time	clear $$ all but router
	Branching	
	flow	clear;default

	enable	break	
	base		
	Disk		
	disk	-1	$$ search all drives for files
	edisk	-1	
	detach	all	
	nsdirwr	-1	$$ always write nameset directory
	Judging		
	enable	arrow	
	okword	'ok'	
	noword	'no'	
	General		
	memory	reset	$$ all but router
	status	reset	
	area	reset	
	flow	reset	$$ all but router
	perm	reset	
	image	reset	
	version		$$ no version emulation
	Videodisc Overlay		
	video	init	
	video	unitinit, on	

intcall

Calls a software interrupt.

intcall	number, pass, receive [;**extended**]	
	number	interrupt routine to call
	pass	data passed to the interrupt routine (8 or 20 bytes)
	receive	data returned from the interrupt routine (8 or 20 bytes)
	extended	forces the use of extended registers

Description

Calls the interrupt specified by *number*. In the non-extended form, the registers AX, BX, CX, and DX are set to contain the four 2-byte values from *pass*. Registers DS and ES are set to point to the global or local variables segment as appropriate for the *pass* variable; SI and DI are set to the offset within the segment. On exit from the routine, *receive* contains the four 2-byte values residing in the AX,BX,CX and DX registers. **zreturn** contains the low byte of the FLAGS register.

The **extended** keyword forces the use of the extended set of registers allocated to the 20 byte *pass* and *receive* variables in the following order:

AX,BX,CX,DX,SI,DI,BP,DS,ES,FLAGS

Addresses

The following system-defined function references are sometimes useful in setting or interpreting registers used with **intcall**:

sysloc(global) Segment address of global variables

sysloc(local) Segment address of the current unit's local variables

varloc(var) Offset address of the variable var

absloc(var) Absolute location of var within memory, expressed as a 4-byte offset with no segment. Corresponds to the segment:offset address as follows:

segment = absloc() ars 4

offset = absloc() $mask$ h0f

absloc = ((segment $mask$ h0000ffff) $cls4$ 4) + (offset $mask$ h0000ffff)

Example

Executes interrupt h12, which returns the memory size. The interrupt requires no input data, so the same variables are used for input and output.

```
define   local
regstrs,8
.        ax, 2              $$AX data
.        bx, 2              $$BX data
.        cx, 2              $$CX data
.        dx, 2              $$DX data
define   end
*  call bios interrupt h12, "memory size determine" no input
*  parameters, so the same variables can be used for
*   input and output
*
intcall h12, regstrs, regstrs
at       5:5
write    You have «s, ax $imul$ 1024» bytes of memory.
```

BNSDOCID: <WO___9907007A2_I_>

System Variable

zreturn contains the lower byte of the computer's internal flags register. The function reference **bit(zreturn,8)** can be used to test the carry (CY) bit, often set by software interrupts to indicate an error condition. See **intcall** online documentation for FLAGS register definition.

jump

Branches to a new main unit.

jump	*UNIT* [(argument/16s)]
jump	*SELECTOR; UNIT*[(argument 16s)]; *UNIT*[(argument *116s*)];...
jumpop	*UNIT*[(argument /16s)]
jumpop	*SELECTOR; UNIT* [(argument/16s)]; *UNIT* [(argument/16s)];...

Description

The **jump** and **jumpop** commands branch to the specified unit making it the new main unit. Optionally, up to 16 arguments may be passed to the new main unit. All **flow** branch settings are cleared (except for **router** settings) and restored to their defaults.

A **jump** branch restores all plotting parameters to their default values (see **status**) and erases the screen (or window) to the **colore** color. All pointer areas are cleared and restored to their defaults.

A **jumpop** (**jump on-page**) branch does not change plotting parameters, **area** settings or perform a screen erase.

The flow command with **jump** keywords performs similar functions. However, **flow** is a delayed event-driven branch while **jump** occurs immediately when the command is executed. **jump** also allows arguments while **flow** does not.

The **SELECTOR** form is described in the "Syntax Conventions". **jump** has an additional **q** (for quit) list entry form that if selected terminates further execution of the current unit. See Example 4.

Up to 16 arguments, separated by commas, can be passed with the branch. These are evaluated with the execution of the **jump** and their values are passed to next unit. A **receive** command in the next unit is used to accept the arguments. See the **receive** command for further information on argument passing and the system variables **zargs** and **zargsin** that are set when passing arguments.

Examples

Example 1

```
jump      two
```

Branches immediately to unit *two* in the current lesson. The screen is erased and all defaults are re-established.

Example 2

```
jump      rivers,«name»
```

Branches to the unit specified by variable *name* in lesson *rivers.*

Example 3

```
jumpop    build (98.6, men+women, size)
...
receive   temp, total, size          $$ in unit build
```

Branches (without a screen erase) to unit *build* in the current lesson passing as arguments: the constant 98.6, the evaluation of the expression *men+women,* and the value in the variable *size.* In unit *build,* the **receive** statement accepts the passed arguments into variables. Since the screen is not erased with **jumpop**, unit *build* can add more to the existing screen display.

Example 4

```
jump     x-2; one; two; three; q
```

This selective form of **jump** branches to unit *one* if *x-2* is negative, to *two* if 0, to *three* if 1 and quits executing any further commands in the current unit for values greater than 1.

Generic Unit Names

Generic branch names are provided for common **jump** destinations to new main units:

=next The unit which physically follows the current main unit in the lesson. If none, the branch is ignored.

=back The unit which physically precedes the current main unit in the lesson. If none, the branch is ignored.

=first The first executable unit in the current lesson.

=last The last executable unit in the current lesson.

=main The current main lesson and unit as held in the system variables **zmainl,zmainu.**

=base The main lesson and unit from which the last **base** modified **flow** branch occurred or as set by the **base** command. If none, the branch is ignored. The names of the current base lesson and unit are held in the system variables **zbasel,zbaseu**. Typically used for returning from a supplementary lesson sequence such as help to the main line of study.

=editor The source editor (loaded with the current executing unit) if executed by the authoring system; ignored by the student executor.

=exit The lesson exit as set by the **exitles** command and contained in the system variables **zexitl,zexitu.** Student users are typically branched back to their routing system such as the TenCORE Activity Manager or DOS

if none is present. An author testing a lesson is returned to the File Directory.

=system Opens the system options window with the calculator, image capture, cursor, etc. [F2] is normally loaded with this branch as a TenCORE system default during authoring; ignored by the student executor.

Example

A learner proceeds to the next main unit if the question is answered correctly on the first try. If two or three tries are required, the current main unit is redone. For any other number of tries, the base unit pointer is set and the student is branched to a helping unit called *trouble*. In unit *trouble*, a **jump** to **=base** would return the learner to the base unit for another try at the question. This code would work equally well as part of the main unit or as a subroutine called by all the questions in the lesson.

```
if      tries =  1
.       jump     =next $$ go on to next question
elseif  tries <  4
.       jump     =main $$ re-do this unit
else
.       base     «zmainu» $$ set base to current main unit
.       jump     trouble  $$ give learner extended help
endif
```

library

Does a subroutine call to a binary unit.

library	*UNIT*[([argument/16s][; return/16s])]
library	*SELECTOR; UNIT*[([argument/16s][; return/16s])];...

BNSDOCID: <WO___9907007A2_I_>

Description

A special form of the **do** command (which see) that does a subroutine call to an external binary library regardless of whether the calling unit is being executed in source or binary mode. The format is identical to **do** including the passing of arguments to and from the called unit. This command is used for calling completed routines or third-party TenCORE software that exits only in binary form. When a binary is made of a lesson, **library** and **do** work identically.

If the *calling* unit is executing in source mode (by running source code directly from the Source Code Editor), **library** switches to binary execution mode for the subroutine call. When returning from the *called* unit back to the *calling* unit, source mode execution is resumed. If the called library routine **jumps** to a new main unit, execution continues in binary mode and the "return to editing" key [F2] no longer operates.

See the **operate** command for a discussion of source and binary execution modes and how they can be explicitly controlled.

loadlib

Loads a binary unit into the unit buffer.

---

loadlib *UNIT*

loadlib *SELECTOR; UNIT ; UNIT;*

Description

Loads a binary unit into the unit buffer cache for execution by a subsequent **library** command. The unit is fetched from a binary file regardless of the type (source or binary) of the invoking unit or the status set by **operate**.

loadlib can be used to check for the existence of a unit before executing it, or to force units into memory. This can be useful for speeding up following sections of code, or for allowing the diskette on which the units reside to be removed.

loadlib should be used with caution: any non-executing unit can be purged from memory if space is needed for the execution of a unit.

Example

Unit *copyfile* is pre-loaded from the binary copy of lesson *libfile*. Unit *copyfile* copies files from one diskette to another. Once loaded, the diskette containing lesson *libfile* can be removed.

```
loadlib  libfile, copyfile
*  force binary unit copyfile into memory
jump     zreturn;; erroru     $$ always check zreturn
at       5:5
write    Now put the diskette to copy FROM in drive A and the diskette to
copy TO in drive B.
         Press Enter to begin the copy
pause    pass = %enter
library  libfile, copyfile    $$ guaranteed to be in memory
```

System Variable

zreturn

-1	Operation Successful
0	Disk error (see zdoserr, zedoserr)
4	Lesson not found
10	Unit not found
11	Invalid type

SNSDOCID: <WO, 9907007A2_I_>

16	Invalid name
21	Conflict with another user's lock

loadu

Loads a unit into the unit buffer.

loadu	*UNIT*
loadu	SELECTOR; *UNIT* ; *UNIT*;...

Description

Loads the specified unit from disk to the unit buffer cache without executing it. **loadu** can be used to load one or more units in advance to avoid later disk access delays, to check if a unit exists, or to load a unit from a diskette that is to be removed.

Example

Unit *copyfile* is pre-loaded. Unit *copyfile* copies files from one diskette to another. Once loaded, the diskette containing the routine can be removed.

```
loadu    copyfile        $$ force unit copyfile into memory
jump     zreturn;; erroru      $$ always check zreturn
at       5:5
write    Now put the disk to copy FROM in drive A and the disk to copy
         TO in drive B.

         Press Enter to begin the copy.
pause    pass = %enter
jump     copyfile        $$ guaranteed to be in memory
```

System Variable

zreturn

-1	Operation Successful
0	Disk error (see zdoserr, zedoserr)
4	Lesson not found
10	Unit not found
11	Invalid type
16	Invalid name
21	Conflict with another user's lock

loop

Starts a loop structure.

loop	[*CONDITION*]	
loop	counter ⇐ start, end [,increment]	
	counter	variable to serve as the index counter
	start	starting value for *counter*
	end	ending value for *counter*
	increment	increment (decrement)for the index counter

Description

Begins a loop structure ended by **endloop** used to repeat a series of commands for a specified number of times or while a condition is true. A **loop** structure can contain **reloop** and **outloop** commands (which see) at the same level of indentation. All other commands must be

indented. Loops may be nested within each other, but the range of the inner loop must be wholly within the range of the outer loop.

The assignment arrow (⇐) distinguishes an ITERATIVE loop from a WHILE loop.

loop [*CONDITION*]

The code within the WHILE loop structure is executed for as long as the *CONDITION* is true. If no tag is given, repetition is continuous unless an exit is provided via **outloop, jump,** etc.

If a condition is present, its value is tested each time execution returns to the **loop** command. If the condition is true, a new iteration is begun; if false, the loop exits.

Example

Plots a line of *X's* on the screen using a WHILE loop.

```
at        5:1
loop      zspace < 80          $$ while zspace is less than 80
.         write X
endloop
```

loop counter ⇐ start, end [,increment]

An ITERATIVE loop repeats for the number of times indicated by the controlling arguments. *counter* is assigned the value of *start.* Each time execution returns to the start of the loop, *increment* is added to *counter.* If *increment* is omitted, the default value is 1. If *increment* is a negative, the value of *counter* will decrease. If *counter* has passed the value of *end* (in whatever direction the loop is counting), the loop terminates.

Example 1

```
loop      index    ⇐ 1,10            $$ loop from 1 to 10
.         at       index:4
.         write    index is «s,index»
endloop
```

Example 2

A WHILE loop is used to limit the horizontal plotting on the screen. The ITERATIVE loop plots a random number of X's with each iteration of the WHILE loop.

```
at       10:1
loop     zspace   <     80          $$ while zspace is less than 80
.        calc           random <= randi(10)
.        loop           index <= 1, random
.        .        write X
.        endloop
.        pause    .5
endloop
```

memory

Manages memory pool data storage.

---

memory keyword;...

create	allocates new memory block
read	transfers data from memory block to variables
write	transfers data from variables to memory block
exchange	exchanges data between memory block and variables
move	transfers data between memory blocks
delete	deletes named memory block
reset	deletes all memory blocks
rename	changes name of memory block
resize	changes size of memory block

BNSDOCID: <WO___9907007A2_I_>

info	returns block information in system variables
total	changes total size of memory pool

Description

Provides access to the memory pool. Memory in the pool is allocated by named blocks. Data in a variable (buffer) can be written and read from these named memory pool blocks. The system automatically manages the memory pool as blocks are created, deleted, and resized. The main memory pool consists of all unused DOS memory below 640 Kilobytes. Secondary(virtual) memory such as expanded(EMS), extended(XMS), and disk memory is used to expand the main memory pool.

Most **memory** command calls should be followed by a **zreturn** check to verify successful performance.

The **initial** command deletes all memory blocks. An execution error also deletes all blocks, and restores the size of the memory pool to the size at startup.

The memory pool is used by the commands: **memory, image, window, status, area, flow, font** and **perm**. Memory blocks are tagged as belonging to a specific command type at creation and cannot be accessed by other commands using the memory pool. Different commands can use the same name for a memory block without conflict.

memory create; 'NAME ',size [; fixed | router | temporary]

Creates a named zero-filled block of the given size in the memory pool. When a block is allocated in the memory pool, its size is rounded up to a multiple of 16 bytes. In addition to its rounded-up length, each memory block uses an additional 16 bytes in the pool.

The optional modifier **fixed** pins the block to a fixed place in memory whose address is found in the system variable **zmstart.** Fixed blocks should be avoided except where absolutely

9NSDOCID: <WO 9907007A2 I >

required since they degrade the operation of the memory pool. An example use of a fixed block is as a communication buffer with an external program.

Normally, a memory block is deleted by **initial** or **memory reset** unless the optional keyword **router** is present A router modified memory block can only be deleted with an explicit **memory delete;** *'NAME'* or **memory reset** ; **router** statement. **router** modified memory blocks are accessible from any domain and are not deleted by the deletion of the domain in which they were created.

A **temporary** modified memory block is deleted automatically when the system requires additional memory pool space: it is useful for holding data that can be regenerated when necessary.

The modifiers are mutually exclusive: only one can occur in a **memory create** statement.

Example 1

Creates a 1000-byte memory block called *temp* in the memory pool. A jump to unit error occurs if an error such as insufficient memory occurs.

```
memory   create; 'temp',1000
jump     zreturn;;error
```

Example 2

Creates a 200-byte **fixed** memory block using the name contained in the 8-byte variable *name.*

```
memory   create; name,200; fixed
jump     zreturn;;error
```

memory write; 'NAME ',offset; buffer; length

Transfers data to the named memory block starting at the given offset from a variable buffer for the given length of bytes. All offsets in memory blocks start at 0: to reference the beginning of a memory block, use an offset of 0.

Example

Transfers 8 bytes from variable *score* to memory block *results*, starting 32 bytes past the beginning (0) of block *results*. Jump to an error routine if something goes wrong.

```
memory   write; 'results',32; score; 8
jump     zreturn;;error
```

memory read; 'NAME ',offset; buffer; length

Transfers data from the named memory block starting at the given offset to a variable buffer for the given length of bytes.

Example 1

Transfers 256 bytes of data from the beginning (offset 0) of memory block *info* to the buffer *infobuf*. Jump to an error routine if something goes wrong.

```
memory   read; 'info',0; infobuf; 256
jump     zreturn;;error
```

Example 2

All global variables are saved and restored over a call to a library subroutine. First, a memory block called *saveVars* is created. The global variables are transferred to this memory pool block. A library routine is called that may change the global variables as desired. Finally, the global variables are restored.

```
define  global
globals=29696     $$ assume default global space (29 Kilobytes)
varBlock,globals,block
define end
*
memory   create; 'savevars',globals          $$ create globals block
jump     zreturn; ; error
memory   write; 'savevars',0; varBlock; globals $$ save globals
```

```
jump      zreturn;;error
...
zero      varBlock,globals          $$ perhaps zero all globals
library routines,graph              $$ can use all globals
...
memory read; 'savevars',0; varBlock; globals $$restore globals
jump      zreturn;;error
```

memory exchange 'NAME ',offset; buffer; length

Exchanges data between the named memory block starting at the given offset, and a variable buffer for the given length in bytes.

Example

```
memory exchange; block,0; buf(i); 12
```

Swaps 12 bytes of data between the start of the memory block named in *block* and the array buffer starting at *buf(i)*.

memory move; 'FROMNAME ',offset; 'TONAME ',offset; length

Moves data directly between memory blocks.

Example

```
memory   move; 'databuf',0; 'backup',3; 256
```

Transfers 256 bytes of data from the beginning of memory block *databuf* to memory block *backup* starting at offset 3.

memory delete; 'NAME'

Deletes a named block from the memory pool.

memory reset [;router]

memory reset deletes all memory blocks except those created with the **router** modifier. Using the optional modifier **router** causes all memory blocks to be deleted including those created with the **router** modifier.

memory rename; 'NAME','NEWNAME'

Renames a memory block

memory resize; 'NAME',size

Changes the size of the named memory block. The memory block cannot be a **fixed** block. If the new size is larger, zeroed bytes are added to the end. if the new size is smaller, bytes are removed from the end. In either case, the remaining data is unchanged.

memory info; 'NAME'

Sets the system variable **zmlength** to the named block's size in bytes (or 0 if the block does not exist). If the block is of **fixed** type, sets the system variable **zmstart** to the block's absolute address in the memory pool. This command can be used to check whether a memory block of a certain name exists.

Example

zreturn checks are used to verify the existence of the memory block whose name is contained in the 8-byte variable *test*. If in the memory pool, its length and address (0 if not of **fixed** type) are displayed.

```
memory   info; test
if       zreturn              $$ test for success (-1)
.        write    length= «s,zmlength» address= «s,zmstart»
elseif   zreturn=10
.        write    Name does not exist
else
.        write  Other Error
endif
```

memory total; size

Changes the size of the memory pool. If the size is increased, additional memory is requested from DOS; if decreased, the unused memory is returned to DOS. By default, all unused DOS memory (below 640K) is allocated to the memory pool; therefore it is not possible to increase the memory pool size unless it has been specifically limited by command line options when starting TenCORE. if a memory-resident program has been loaded after TenCORE, the memory pool size cannot be increased.

Secondary Memory Pool

When space runs out in the memory pool, existing blocks in the pool are automatically moved to the secondary memory pool to make room for new blocks. The secondary memory pool can be any or all of the following: expanded (EMS) memory, extended (XMS) memory, or disk. If more than one memory type is available, first expanded (EMS), then extended (XMS), and finally disk is used.

When starting TenCORE, a default amount of available EMS, XMS, and disk space is automatically allocated to the secondary memory pool. If desired, the amount of each can be set upon system startup.

When a secondary memory pool is available, the maximum size of a block is still limited by the size of the main memory pool; however, more memory pool blocks can be created since existing blocks can be moved to the secondary memory pool.

Creating **fixed** memory pool blocks limits the maximum size of further memory pool blocks because they cannot be moved to secondary memory. In general, use of **fixed** memory blocks is not recommended.

System Variables

zreturn

The **memory** command sets **zreturn** as follows:

1	Operation successful
3	Variable reference out of range
6	Duplicate name
10	Name does not exist
11	Invalid type (resize of fixed block)
16	Invalid name
18	Insufficient memory in pool (create and resize keywords) or insufficient DOS memory to add to pool (total keyword)

Miscellaneous

The following system variables are set after any **memory** command which references a memory block name (except **delete**):

zmlength	Size, in bytes, of last referenced memory block
zmstart	Absolute address of last referenced memory block if fixed; otherwise 1

The following system variables provide information about the TenCORE memory pool:

zrmem	Largest possible size for a new memory block that is not **fixed**
zfmem	Largest possible size for a new **fixed** memory block
zmem	Size of memory pool
zxmem	Amount of extra DOS memory which could be added to the memory pool (with **memory total**)

mode

Controls how graphics, text and images are plotted on the screen.

mode	*mode*
mode	*SELECTOR; mode; mode;...*
write	«**mode** | **m**, *mode*»

The *mode* plotting keywords are:

write	use foreground color
erase	use background (erase) color
rewrite	use foreground color, erase character background
inverse	use background (erase) color, plot character background
noplot	simulates plotting without affecting the display
xor	use foreground color with *EXCLUSIVE OR* on existing display
add	use foreground color with *OR* on existing display
sub	use inverted foreground color with *AND* on existing display

Description

Controls how graphics, characters and images are put on the display screen. Each of these three types of objects has its own structure and interaction with the mode and color commands. Graphic objects such as **draw** and **circle** consist of the dots defining the object; they use the foreground color when plotting on the screen. Characters consist of foreground dots that define a character and the remaining background dots of the character cell: the foreground and background dots perform different tasks in the various modes using the foreground and erase colors. Images consist of a rectangular area of already colored dots and do not use the

foreground or background colors. However, a dot of color black (palette slot 0) in an image can be used for transparency to any pre-existing display.

For text-only display screens, individual dots are not addressable: only whole character positions can be changed which limits the functionality of many of the modes. Text-only mode operations are discussed separately at the end.

The mode is reset to the default mode upon a **jump** branch to a unit. The initial default setting for mode is **write**. This can be changed through use of the status **save** statement.

A change in mode through an embedded **mode** command in a **write** statement does not extend beyond the **write**: upon finishing a write, the mode is restored to its value as at the start of the **write**.

mode write

Plots graphics and character dots in the foreground color. Images are plotted with their designed colors. Character backgrounds and the slot 0 color of images are transparent to any existing display.

mode erase

Erases graphics and character dots to the background (erase) color. Images are erased as for mode sub.

mode rewrite

Plots graphics and character dots in the foreground color. Character backgrounds are erased to the background (erase) color. Images are plotted entirely with their designed colors including the slot 0 color.

Example

A "count-down" is performed counting down from 10 to 0 followed by a "BANG!".

```
mode      rewrite
```

BNSDOCID: <WO_ _9907007A2 I >

```
loop    count <= 10,0,-1
.       at 10:25
.       showt   count,2
.       pause   1
endloop
at      10:25
write   BANG!
```

mode inverse

Plots graphics and character dots in the background (erase) color. Character background is plotted in the foreground color. Images first have all colors inverted to their bit-wise complements and then are plotted entirely including color slot 0.

mode noplot

Simulates plotting without altering the screen. The screen location and all plotting attributes except **mode** are updated as though real plotting had occurred.

Example

The length of a line of text is first determined using **mode noplot** and put into the variable *length*. This is done by setting the screen x position to 0 by an **at**, plotting the text in **mode** noplot, then getting the ending position found in the system variable **zx**. The text is then really plotted centered on the screen or display window. This is done in the last **at** statement by halving the display width (**zxmax** /2) then moving back by half the text width *(length/* 2).

```
mode    noplot$$            do not affect screen
at      0,100               $$ start at x=0
write   Handling Snakes     $$ simulate plotting of text
calc    length <= zx        $$ the system variable zx gives
*                           $$ length of text
mode    write               $$ set to standard plotting mode
```

BNSDOCID: <WO__9907007A2_I_>

```
at      zxmax/2 - length/2,100     $$ set start of text
*                                  $$ for centering
write   Handling Snakes
```

mode xor

Graphics, character dots and images are logically *EXCLUSIVE OR* combined with any existing display. Plotting the same material twice on the screen in the same location in this mode returns the display to its original composition.

Example

The word "ORBIT" is made to animatedly rotate around the center of the screen. If "ORBIT" runs through some background graphics, it would do so without altering the background. *rad* is defined as a real variable and holds the radian for the sin/cos calculations.

```
screen   vga
calc     rad <= 0
mode     xor                  $$ switch to EXCLUSIVE OR plotting
loop
.        at       sin(rad)*100 + 320, cos(rad)*100 + 240
.        write          ORBIT
.        pause          .01
.        at       sin(rad)*100 + 320, cos(rad)*100 + 240
.        write          ORBIT
.        calc           rad <= rad + .05
endloop
```

mode add

Graphics, character dots and images are logically *OR* combined with any existing display.

mode sub

Graphics, character dots and images are color inverted and then logically *AND* combined with any existing display.

Text-Only Hardware Screens

The modes **write, erase, rewrite** and **inverse** work as expected changing the character foreground and background colors as appropriate. The logical modes **xor, add** and **sub** change the foreground character color as explained already. However, xor plots characters only where blank (space) characters already exist on the screen and otherwise erases characters from the screen.

System Variable

zmode	Holds the current plotting mode	
	0	inverse
	1	rewrite
	2	erase
	3	write
	4	noplot
	5	xor
	6	add
	7	sub

move

Moves bytes of data from a literal or variable into a variable.

move from, to, length

BNSDOCID: <WO___9907007A2_I_>

Description

Moves the specified number of bytes from one location to another. if both *from* and *to* are variables, they may overlap, and the length of data moved may exceed the defined length of either variable. The **move** command is extremely fast.

Example

Moves *rec(2)* through *rec(5)* down one record.

```
define   local
reclen = 256              $$ TenCORE data record is ALWAYS 256 bytes
rec(6),reclen                          $$ array of six records
define   end
*
move     rec(2), rec(3), 4*reclen   $$ move the records down
```

nextkey

Advances system key input buffer and variables.

nextkey [keyword]

blank	advances key buffer and updates **zinput**
test	updates **zinput** without advancing key buffer clear
clears	the key input buffer
flags	updates **zinputf** and **zkeyset**
pointer	updates **zinputx, zinputy** and **zinputf,** sets zinput to %ptrinfo

Description

nextkey reads pending keys from the system key input buffer. Input is normally buffered until the end of a main unit, a **pause**, an **arrow**, or a **no** judgment is encountered. **nextkey** checks

the input buffer immediately, without waiting for one of these waiting states. The blank tag form copies the next available key from the buffer into **zinput** and removes the key from the buffer

If the key buffer is empty, the system variable **zinput** is set to zero.

Encountering a **nextkey** does not activate current **flow** settings.

nextkey

Updates **zinput**, **zinputf**, and **zkeyset** with the next available key in the input buffer, removing the pending key from the buffer. If the pointer is enabled, **zinputx** and **zinputy** are also updated.

nextkey test

Updates **zinput**, **zinputf**; and **zkeyset** like the blank-tag form, but does not remove the pending key from the key buffer

nextkey clear

Clears the input buffer of all pending keys. Does not clear **zinput**.

nextkey flags

Updates **zinputf** and the left-most byte of **zkeyset**, and zeroes the right-most byte of **zkeyset** No change is made to **zinputx** and **zinputy**.

nextkey pointer

Updates **zinutx**, **zinputy** and **zinputf**. Sets **zinput** to **%ptrinfo**.

Example

The question "What goes meow?" is shown as an animated billboard at the top of the display. When the user presses a letter on the keyboard, the animation stops and the letter appears at the following arrow. The variable x is defined as a 2-byte integer.

```
margin    wrap                $$ wrap around screen
mode      rewrite             $$ replace any underlying text
```

ENSDOCID: <WO__9907007A2_I_>

```
calc     x <= zxmax / 2        $$ start in middle of screen
nextkey  clear                 $$ clear any pending keys
nextkey                        $$ clear zinput also
*
loop     zinput = 0            $$ loop while no input keys
.        at       x,zymax-20         $$ near top of screen
.        write    What goes meow? $$$ text will move to left in loop
.        calc     x <= x - 1   $$ move starting x to left 1 pixel
.        calcs    x < 0; x <= zmax;;$$ check for screen wrap
.        nextkey  test          $$ update zinput
endloop
*
arrow    10,zymax-40
answer   cat
endarrow
```

System Variables

zinput	0 no input
	1-255 character value of last keypress
	>255 non-character keypress
zinputf	Flags for the input in **zinput**
zkeyset	Flags and scan code for the input in **zinput**
zinputx	X-coordinate of the pointer device for the input in **zinput**
zinputy	Y-coordinate of the pointer device for the input in **zinput**

nocheck

Cancels subroutine call argument count checking.

nocheck return | receive	
nocheck return , receive	
return	Turns off checking of argument counts for **return** commands
receive	Turns off argument count checking for **receive** commands

Description

Disables the automatic checking which is done on **return** and/or **receive** commands to insure that the correct number of arguments is being passed.

nocheck allows any number of tags to be present in **return** or **receive**, regardless of the number of arguments specified by the *calling* (**do**, **library**, or **jump**) command.

nocheck must be in the same unit as the **return** and **receive** commands to be affected (the *called* unit).

operate

Switches between different execution modes.

operate	mode
operate	*SELECTOR;* mode; mode;..
The *mode* keywords are	
source	execute subsequent units from *.SRC* files
binary	execute subsequent units from *BIN* files
tpr	execute subsequent units from *.TPR* files

BNSDOCID: <WO__9907007A2 I_>

Description

Sets the mode of operation, **source, binary** or **tpr**, for system file execution. When a command executes which requires the system to access a unit, the mode of operation determines whether it looks for the unit in a **source** *(.SRC)*, **binary** *(.BIN)* or **producer** *(.TPR)* file. The default is to access the same type of file as the current unit was loaded from.

The **operate** command remains in effect until it is changed by another **operate** command, a **library** command, an **error** branch, or execution of a **flow** with an **operate** tag.

If the mode is changed by a **flow do,** it is restored upon completion of the **do** sequence. If the mode is changed by **flow jump**, it is not restored.

Executing **operate source** in the student executor generates execution error 11,"invalid argument value".

error saves the mode of operation at the time the command is executed (at the time the branch pointer is loaded). When the **error** branch is taken, the mode of operation is reset to the mode which was in effect at the time the pointer was loaded.

Interaction with library Command

Execution of **library** temporarily sets the mode of operation to binary. if execution returns normally from the binary unit, the original mode of operation is restored.

However, two things can prevent restoration of the original mode of operation:

- entering a new main unit from a library unit, in which case no return from **library** occurs and binary operation continues.
- executing an **operate** while in a library unit, in which case no restoration of the mode of operation is performed on return to the calling unit. If the calling unit was itself called by a **library**, the mode of operation is restored on return to that calling unit.

origin

Sets the origin for the display coordinate system.

---

origin [*LOCATION*]

Description

Moves the origin for the coordinate system from the default location (0,0) to the specified location. All subsequent display commands are positioned relative to the specified origin.

The origin is set to (0,0) at the start of each new main unit. This can be changed using **origin** followed by **status save; default.**

A blank-tag resets the origin to 0,0

Example

Displays a box inscribed with an x on three different areas of the screen.

origin

```
origin   120,70   $$ set origin for first box and x
do       xbox     $$ plot it relocated
origin   160,100  $$ second box
do       xbox
origin   200,130  $$ third box
do       xbox
```

xbox

```
*          this unit draws a 4-pixel thick box with an "x" inside it
*          it is normally plotted in the lower left corner of the
*          display
*
box      0,0;20,20;4          $$ draw the box
```

BNSDOCID: <WO___9907007A2_I_>

```
draw      0,0;20,20;;0,20;20,0 $$ followed by x
```

System Variables

zoriginx	Absolute X-coordinate for the current origin
zoriginy	Absolute Y-coordinate for the current origin

outloop

Exits a loop structure from any point within the structure.

---

outloop [*CONDITION*]

Description

Provides an exit from any point within the loop structure. If the tag is blank or the condition is true, **outloop** goes to the command following the associated **endloop.** If the condition is false, no action is taken and execution continues with the next command within the loop structure.

outloop always appears at the same indentation level as the loop structure to be exited, regardless of any other indented structures surrounding the **outloop**.

Example

```
define    local
counter,    2          $$ counter
define    end
local
*
at        5:3
write     Press Enter to increment the counter, or
              Esc to exit the loop.
calc      counter <= 1
```

```
loop
.       at  8:3
.       erase     50
.       write     The counter value is «show, counter».
.       pause     pass= %escape, %enter
outloop zinput = %escape     $$ exit if Esc is pressed
.       calc      counter ⇐ counter + 1
endloop
*
at      10:3
write   You exited the loop!
```

pack, packz

packc, packzc

Places text string into buffer.

pack	buffer; [length]; *TEXT*
packz	buffer; [length]; *TEXT*
packc	*SELECTOR;* buffer; [length]; *TEXT; TEXT;...*
packzc	*SELECTOR;* buffer; [length]; *TEXT; TEXT;...*
buffer	A variable into which a text string is placed.
length	An optional variable that is assigned the number of characters placed into the buffer
TEXT	The text string to be placed into the buffer.
SELECTOR	An expression used for selecting one of the *TEXT* cases.

BNSDOCID: <WO__9907007A2_I >

Description

Places a text string into the given buffer and if a length variable is specified, returns the number of characters actually stored. If the defined buffer is longer than the text string, the **pack** form leaves the extra bytes unchanged while the **packz** form zeroes them. A **c** (conditional) suffix on the command denotes the *SELECTOR* form.

The text string can contain embedded **show**-type commands such as **show, showt,** and **showa**. If the text extends beyond one line, a carriage return (code 13) is packed into the buffer before each additional line. If the text string continues over blank lines, multiple carriage returns are included. If the length of the text string is not required, use a null argument for length.

The **pack** commands do not perform bounds checking: characters being packed can overrun a buffer and overwrite subsequent variables or cause an execution error if they extend beyond defines. The maximum possible length of a text string must be allowed for.

***SELECTOR* Delimiters**

The delimiter following the length argument is used by **packc** and **packzc** to delimit the text strings for each case. The following delimiters are valid:

comma ,

semicolon ;

colon :

end-of-line ↵

universal delimiter |

Two consecutive delimiters indicates a null tag; nothing is packed into the buffer for the corresponding value of *SELECTOR*. If a delimiter other than end-of a line is being used, a single string can span more than one line, in which case a carriage return is packed in for each end-of-line encountered.

Null Codes

Any null characters (code 0) in an embedded **showa** command are skipped; the null characters do not appear in the resulting string. Null characters can be forced into a packed string by using an embedded **showv** command with a variable that contains the null characters.

Examples

Example 1

Puts three lines of text into the buffer *lines.* They are separated by carriage return codes in the buffer. *count* is set to the length of the string packed. The **showa** statement would display the three line text string.

```
pack    lines;count;The first line of text.
        The second line of text.
        The last line of text.
showa   lines,count
```

Example 2

Embedded commands are used to include the name in buffer *name* and integer in variable *age* within the text string being packed into buffer *introbuf* Any extra space in *introbuf is* zero filled. No length argument is used. The **showa** statement would display a message such as "My name is Alex. I am 23 years old." Any following zero codes in *introbuf are* skipped by the system plotting routines.

```
packz   introbuf;;My name is «a,name». I am «s,age» years old.
showa   introbuf
```

Example 3

Based on variable *number,* the text *negative, zero, or positive* is conditionally placed into *textBuf* setting *count* to the number of characters transferred. *End-of-line* is used as the *SELECTOR* delimiter. Unused space in *textBuf* is zero filled.

```
packzc   number;textBuf;count
         negative
         zero
         positive
```

Example 4

The buffer *textDate* is set to a character string with today's date (for example "January 1, 1993"). The variable *temp* is used to hold the month until it is packed into *textDate* along with the day and year. Note the three null arguments on the **packzc** command: the first indicates no length information is required; the next two handle the *SELECTOR* negative and zero cases which are not possible with *month*.

```
define   local
today,   4
.        year,    2
.        month,   1
.        day,     1
temp,    9
textDate,20
define   end
*
date     today          $$ obtain the system date information
*
packzc   month;temp;;;;January;February;March;April;May;June;July;August;
         September;October;November;December
*
packz    textDate;; «a,temp» «s,day», «s,year»
*
at       5:10
```

```
showa       textDate    $$ display today's date
```

page

Selects the hardware screen page to use for plotting and display.

page	showpage [,writepage]	
	showpage	page number to display
	writepage	page to plot on

Description

Some display hardware has several separate, complete displays called *pages.* The **page** command is used to control them.

The one argument form of **page** changes the active display page to the one specified. The contents of that page are displayed immediately and all further plotting occurs on that page.

The two argument form causes the page given first to be displayed while the page given second is used for all further plotting activity. This allows the program to construct a new, hidden page in memory while the user is viewing the displayed page.

Entering a new main unit erases only the page selected for plotting.

The number of pages available depends on the display hardware being used, the display memory available, and the screen resolution selected. The system variable **zmaxpage** contains the number of pages that the current screen supports.

Example

Animates an image named *fish* in **mode xor.** Because redisplaying an object in **mode xor** removes it cleanly from the display, mode **xor** is used when an animated object must pass over other objects without destroying them. Without **page**, **mode xor** animations usually show too much flicker to be useful.

Page 1 is used to show the image at odd x-locations 201, 203, 205, etc.

Page 2 is used to show the image at even x-locations 202, 204, 206, etc. By alternating pages, the image is made to appear at 201, 202,203, 204, 205, 206, etc.

```
screen   ega, graphics,medium,color
define   local
x, 2
define   end
* set mode and load image into memory
mode     xor
image    move; block, 'fish'; memory, 'fish'
* display the first image on page 1
image    plot; memory, 'fish'; 201,100
* while displaying page 1,
* erase page 2 and put the next image on it
page     1, 2
erase
image    plot; memory, 'fish'; 202,100
loop      x <= 201,400
.           * show page 2 while changing the display on page 1
.                 * ...re-display (thus removing) the old fish on page 1
.                 * ...and display a new fish 2 pixels to the right on page 1
.           * then increment x-location counter by 1
.           page  2,1
.           image plot; memory, 'fish'; x,100
.           image plot; memory, 'fish'; x+2,100
.           calc  x <= x + 1
.           *     keep page 1 showing while changing the display on page 2
.           page  1, 2
```

```
.           image plot; memory, 'fish'; x,100
.           image plot; memory, 'fish'; x+2,100
endloop
```

System Variables

zrpage	Page currently displayed
zwpage	Page on which text and graphics is being written
zmaxpage	Maximum number of pages available for screen type in effect

palette

Changes the palette.

---

palette *keyword*

init	initializes the palette to the default state
set	sets color values for individual palette entries
read	reads palette entries from hardware to variables
write	writes palette entries from variables to hardware
color	associates a color keyword with a palette entry

Description

Changes the set of colors available on palette-based display hardware: EGA, MCGA, VGA, EVGA and compatible graphics adapters. **palette** supports several different functions:

- Changing all or a portion of the current palette
- Reading all or a portion of the current palette setting into variables.
- Changing a color keyword to correspond to a different palette entry.

A "palette" is the set of colors assigned to the hardware color slots. VGA screens have 16 color slots, and MCGA and EVGA screens have 256 color slots. The "standard palette" is the set of default values for those slots.

Each color is a combination of three color values: red, green, and blue. Together, they are referred to as RGB. Some modes also use an intensity byte to indicate relative brightness for each color. For TenCORE, the standard palette uses the following color assignments:

0	black	4	red	8	black+	12	red+
1	blue	5	magenta	9	blue+	13	magenta+
2	green	6	brown	10	green+	14	brown+
3	cyan	7	white	11	cyan+	15	white+

There is no inherent connection between a slot number and the RGB values that it contains. This is relevant when displaying more than one image on the same screen. If the images use different palettes, only the palette of the most recently displayed image will be active. Since the same slots are in use by the other images, their colors might change when a different palette becomes active. Careful planning of a common palette for multiple images on a screen is needed.

The default palette is restored whenever a new main unit is entered. The default palette can be altered by executing a **status save** with the **palette** modifier after changing the palette. The palette is reset to the standard default by executing a **status restore;standard;palette**.

palette init **[;number]**

Initializes the palette to its standard default state. On cards that support multiple defaults, *number* specifies the desired default palette.

palette set; COLOR, red, green, blue, [,intensity]

Sets the values of red, green, blue, and optional intensity for the palette slot referred to by *COLOR*. Palette slots can be specified by number or color keyword.

COLOR is any hardware color number valid for the hardware in use. *red, green, blue* and *intensity* range from 0 to 255. For hardware which does not support so wide a range, the values are automatically scaled: 255=full intensity, 128=half, etc. The system variable **zpalincr** reports the minimum significant increment for the color values; **zintincr** gives the minimum significant increment for intensity.

intensity is significant only for two screen types: **ega,graphics,low** and **ega,graphics,medium**. On other screen types, **zintincr** is 0 and the intensity value is ignored. This can cause problems when a palette originally generated on either of these two screen types is used on another screen type: it can result in a "dimming" of the palette on the new screen type. This is because the intensity is no longer available, and must be corrected by adjusting the RGB values of the palette.

palette read; buffer, number

Reads the requested number of palette entries into the variable *buffer*. The structure of the buffer is explained below.

palette write; buffer, number

Writes the requested number of palette entries from a variable *buffer*. The structure of *buffer* is explained below.

palette color; COLOR, slot

Associates a TenCORE color keyword with a palette slot. The TenCORE color names by default correspond to the first 16 palette slots. The **color** option allows the author to change which palette slot corresponds to a given color name.

Palette Variables

The **read** and **write** keywords allow palette slot entries to be read to or written from a buffer. The palette buffer contains one or more 6-byte palette entries of the form:

BNSDOCID: <WO___9907007A2_I_>

slot (2 bytes)

red (1 byte)

green (1 byte)

blue (1 byte)

intensity(1 byte)

The value of *slot* must be set for each palette entry before executing the **read** or **write**. Any subset of palette entries may be read or written by setting appropriate *slot* values.

Example

To read the palette information for the default 16 TenCORE color slots:

```
define   local
palette(16), 6
.        slot, 2
.        red, 1
.        green, 1
.        blue, 1
.        intensty, 1
count    , 2
define   end
*
loop     count <= 1, 16
*         must remember to initialize slot numbers
.        calc     slot(count) <= count - 1
endloop
*
palette read; palette(1), 16
```

System Variables

zpalincr Minimum significant increment in color values for current screen type

zintincr Minimum significant increment in intensity values for current screen type

pause

Pauses execution until event occurs.

---

pause [time;] [**flow= all** | *KEYs;*] [**pass=all** | *KEYs*]

pause [time;] **testkeys= all** | *KEYs*

time delay time in seconds before continuing execution

flow= key list for **flow** branching

pass= key list to continue execution

testkeys= like **pass=** but key not removed from input buffer

---

Description

Stops execution of commands until an accepted event, such as a keypress, occurs. The keypress can pass the pause and continue or perform **flow** defined key branching. When a keypress is handled by **pause** it is normally removed from the input buffer and the system input variables such as **zinput** are updated to reflect the new event.

Any **break** modified **flow** branch always works: it interrupts any **pause** to perform its task. Other **flow** branches are active at a **pause** only if their keys are specified in the **flow=** keylist. In either case, if the **flow** is a **do** or **library** branch, control returns to the **pause** when the subroutine ends.

If a time is specified, it sets a clock running that issues the default **%timeup** key when the time expires. This **%timeup** key is automatically included in the **pass=** keylist; unless some other

accepted event occurs first, the **%timeup** allows execution to continue. If a **flow do** or **flow library** branch occurs during a timed **pause**, timing is suspended until the subroutine returns control to the **pause**, then timing resumes.

A **pause** command can extend over several lines if needed to list all options; a blank command field signifies continuation.

pause

A **pause** command with no tag waits until any event occurs at which time command execution continues. It provides for a temporary program halt: the user presses any key and the program continues. Only **break** modified **flow** branches alter the inevitability of executing the next command.

Example

```
write    Press Any Key to Continue
pause
write    Good, now let's open the valve and see what happens...
```

pause time

A **pause** command with only timing forces a delay for the specified time. Only **break** modified **flow** commands can interrupt the time delay. Any other events are ignored and removed from the key input buffer. For a forced delay that doesn't empty the input buffer of events occurring during the delay, use the **delay** command.

There is only one default timer key used for both the **pause** and **time** commands: **%timeup**. Timing in the **pause** supersedes any default time set by a previous **time** command.

Example 1

```
at       5:10
write    Some text...
pause    5
```

```
at        7:10
write     Some more text...
```

After a 5 second delay, additional text is added to the display.

Example 2

```
flow      do; %f5; data; break
...
time      30
...
pause     5
```

The program is paused for 5 seconds: the previous timing (**time 30**) is cancelled. If [F5] is pressed, the timing is suspended while the **break** modified **flow** branch to unit *data* is done. Upon return, the **pause** waits for the remaining time.

pause [time ;] flow= all | KEY/*s*

A **pause** with a **flow=** keylist allows only the specified or **all** flow branches to work. If a **do** or **library** branch occurs, control returns to the **pause** command when the subroutine ends. Optionally, timing can be used to end the **pause** after a given number of seconds. **break** modified **flow** branches always take precedence, whether or not they are listed.

Example 1

```
pause     flow=all
```

The program is paused until any flow branch occurs. Coding following the **pause** will never be executed.

Example 2

```
flow      jump; %f1; out
flow      do; %f2; info
flow      do; %f3; data
...
```

BNSDOCID: <WO__9907007A2_I_>

```
pause    10; flow= %f1, %f2
```

The program is paused for 10 seconds unless [F1] is pressed to **jump** to unit *out* If [F2] is pressed, timing is suspended while unit *info* is done as a subroutine. Upon return from unit *info*, the **pause** waits for the remaining time. Unit *data* is not accessible from this pause since its branch key is not in the **flow=** keylist.

pause [time ;] pass= all | KEY/s

A **pause** with a **pass=** keylist allows the specified or **all** keys to pass the **pause** to continue execution of the following commands. Optionally, timing can be used to end the **pause** after a given number of seconds; the resulting **%timeup** key is an automatic entry in the **pass=** keylist. Only **break** modified **flow** branches work here and take precedence over any keys in the **pass=** keylist.

Example 1

A choice page allows the user to choose a subject for study. Upon choosing one of the options at the **pause**, the program continues to branch (with arguments) to the appropriate section of the lesson. The **break** modified branch to unit *index* is the only **flow** branch which can occur at the **pause**.

```
flow     jump; %f1; index; break
flow     jump; %f2; notes
...
write    Choose an option:
         a. anthracite coal
         b. bituminous coal
         c. peat
pause    pass= a, b, c          $$ only accept a,b,c
if       zinput = "a"
```

```
.       jump     unita(3,5,1)
elseif  zinput = "b"          $$ et cetera
...
```

Example 2

```
pause   10,pass=a,b,c,d,e,f,g,h,i,j,k,l,m,
        n,o,p,q,r,s,t,u,v,w,x,y,z,
        0,1,2,3,4,5,6,7,8,9
```

The tag of a **pause** command can extend over several lines. This **pause** waits for 10 seconds or until one of the listed keys is pressed.

Example 3

```
write   Lots and lots of text
        over many,
        many lines.
nextkey clear
pause   pass=a,b,c
```

If there is time consuming coding such as a large display, keys might be stacked up in the input buffer before a **pause** command is executed. The **pause** processes these waiting keys until it comes to one that it passes. This can be desirable sometimes since it allows the user to type ahead. If type ahead is not desired, use a **nextkey clear** statement before the **pause**. It empties the key buffer of all pending keys.

pause [time ;] flow= all | KEY/s ; pass= all | KEY/s

Both the **flow=** and **pass=** keylists can occur in the same statement with **pass=** keys taking precedence over **flow=** keys. Optionally, timing can be used to end the **pause** after a given number of seconds. **break** modified **flow** branch keys work in all instances.

BNSDOCID: <WO__9907007A2_I_>

The pseudo-key value **%other** can be placed in either keylist to include all keys that are not individually assigned. If put into the **flow=** keylist (or by use of the **all** keyword), then any **flow** statement with the **%other** pseudo-key is activated.

Example 1

```
pause    flow=all; pass=%other
```

All **flow** keys are active. All other keys pass the **pause** to continue execution of the unit.

Example 2

```
flow     jump; %timeup; timeout
...
time     10
pause    flow=all; pass=a,b,c
```

Timing here has been set before and not overridden by the **pause** statement. When the system generates a **%timeup** key, a **flow** branch to unit *timeout* occurs.

Example 3

```
flow     jump; %f1; =exit
flow     jump; %f2; index
flow     do; %f5; data
flow     jump; %timeup; timeout
...
pause    10; flow=all; pass=a,b,c,%pointer
```

Pressing [A],[B] or [C] or any pointer click passes the pause to the following command. Any of the specified **flow** keys (but %timeup) perform their defined **flow** functions. If no other accepted event occurs for 10 seconds, execution continues with the command following the **pause**. Control does not pass to unit *timeout* since the **pause** command's timing automatically includes **%timeup** in the **pass=** keylist, which takes precedence over **flow** assignments.

BNSDOCID: <WO_ _9907007A2_I_>

Example 4

```
flow      jump; %f1; =exit
flow      jump; %f2; index
flow      do; %pgup,%pgdn; nextunit
flow      do; %other; badKeys
...
pause     flow=all; pass=a,b,c,%space,%f2
```

Pressing [A],[B],[C] [] or [F2] passes the **pause** to the following command. The %f2 **flow** branch is not taken because the key is in the **pass=** keylist which takes precedence. The %fl **flow** branch and any other **flow** branches work because of the **flow=all**. Any key not included in the **pass=** keylist and not defined in other **flow** statements performs the **flow do** to unit *badKeys* which handles unanticipated keys.

pause [time ;] testkeys= all | KEY/s

The **testkeys=** form is similar to **pass=** except that passed keys are not removed from the input buffer. It is used to test pending input since the system input variables such as **zinput** are updated. The passed key remains in the input buffer until a following input processing point (**pause**, **arrow**, **nextkey**, end-of-unit). **testkeys=** is the **pause** form of the **nextkey test** command.

testkeys= cannot occur with any other of the keylist forms.

Example

```
pause     testkeys=0,1,2,3,4,5,6,7,8,9
long      1,judge
arrow     10:10
answer    3
.         write    Great!
wrong     2
```

BNSDOCID: <WO_9907007A2 I >

```
.       write   No, that's Mars.
wrong   1
.       write   No, that's Mercury.
...
endarrow
```

Only the number keys cause termination of the **pause**. Since the passed key is not removed from the input buffer by the **testkeys=** form, it remains as input for the following **arrow.** After displaying on the screen at the **arrow**, the key immediately causes judging to occur due to the **long 1, judge.**

Obsolete Form

The ambiguous **keys=** form of the previous version of the system is now replaced with the explicit **pass=** and **flow=** keylist forms. It was not possible by looking only at the **keys=** keylist to predict whether a listed key would perform a **flow** branch or pass the **pause** to the next command. While the **keys=** form still works for reasons of upward compatibility, users are advised to switch to the new unambiguous forms.

System Variables

The input-related system variables are updated whenever **pause** processes a key.

zinput	Most recent keypress, pointer input, or pseudo-key
zinputa	Area ID causing current zinput value
zinputf	Bitmap of keyboard conditions at time of event
zinputx	Pointer x-coordinate at time of event
zinputy	Pointer y-coordinate at time of event
zkeyset	Keyboard scan code
zpcolor	Color of screen under pointer at time of event

3NSDOCID <WO 9907007A2 I >

perm

Manages permutation lists.

perm	keyword...	
	create	generates permutation list in memory
	next	gets next number from permutation list
	remove	removes specific number from permutation list
	replace	puts specific number back into permutation list
	copy	makes copy of permutation list
	delete	deletes permutation list from memory
	reset	deletes all permutation lists from memory
	read	reads permutation list into variables
	write	writes list in variables to permutation list

Description

The permutation managing options available through the **perm** command allow for the random selection, removal and replacement of a number from a list of numbers. Multiple permutation lists can be accessed simultaneously. Copies of a list can be saved and restored at any time in memory or even to the disk between sessions to allow for any possible strategy in using permutations.

Permutation lists can be created in the one system **default** buffer or in any number of named buffers. If possible, the use of the **default** buffer simplifies using permutations since the name argument can be eliminated. The **default** buffer for the **perm** command is shared by all units.

BNSDOCID: <WO__9907007A2_I_>

Permutation list buffers are kept as memory blocks in the memory pool. The memory pool is used by the commands: **memory, image, window, status, area, flow, font** and **perm.** Memory blocks are tagged as belonging to a specific command type at creation and cannot be accessed by other commands using the memory pool; different commands can use the same name for a memory block without conflict.

perm create; length [; 'NAME' | default]

Creates a permutation list of non-repeating, randomly-ordered integers from 1 to the specified length (maximum 32,767). The list is stored as the **default** or a user named buffer in the memory pool. Creating a new **default** list automatically supersedes any previous use of the **default** buffer. Creating a named list produces an error if the named list already exists; use the **delete** option first if you wish to use the same name again.

Example 1

Creates a default permutation list of integers from 1 to 10.

```
perm     create; 10
```

Example 2

Creates a permutation of integers from 1 to 25; the list is saved in the memory block named *quiz 1.*

```
perm     create; 25; 'quiz1'
```

perm next; variable [; 'NAME' | default]

Randomly selects a permutation number and stores it in a variable. The number selected is removed from the permutation list If the permutation list is empty or an error occurs, 0 is returned.

Example 1

Gets the next number from the default permutation list and puts it into the variable *question;* the number is removed from the list.

```
perm     next; question
```

Example 2

Like Example 1 but the number comes from the named list *quiz1*

```
perm     next; question; 'quiz1'
```

Example 3

A permutation of 5 items is created in the default buffer. Within a loop, the five questions in units *Q1* through *Q5* are presented in random order. When the list is exhausted, the 0 value in variable *question* is used to exit the loop structure.

```
perm     create; 5                          $$ create default permutation list
jump     zreturn; ; error
loop
.        perm     next; question            $$ get next permutation number
.        do question;;;Q1;Q2;Q3;Q4;Q5       $$ call question
outloop  question = 0                       $$ exit loop if done
endloop
```

perm remove; itemNumber [; 'NAME' | default]

Removes the specified item from a permutation list. The integer is no longer available for selection by a **perm next** statement.

Example

Removes the value in *problem* from the default permutation list.

```
perm     remove; problem
```

BNSDOCID: <WO___9907007A2_I_>

perm replace; itemNumber [; 'NAME' | default]

Puts the specified item back into a permutation list. The integer is available again for selection by a **perm next** statement.

Example

Puts the value in *problem* back into the default permutation list

```
perm     replace; problem
```

perm copy; ['FROM' | default] ; ['TO' | default]

Makes a copy of a permutation list. If the copied-to list already exists, it is overwritten. The **default** list can occur only once in the statement.

Example 1

Copies permutation *list first* into the named list *second.*

```
perm     copy; 'first'; 'second'
```

Example 2

Copies the default permutation list to the list named in the variable *newList*

```
perm     copy; default; newList
```

perm delete [; 'NAME' | default]

Deletes a permutation list from memory.

perm reset

Deletes all permutation lists from memory.

perm read; permBuffer [,length] [; 'NAME' | default]

Reads a permutation list into the specified buffer for a given length in bytes. If length is not specified, the defined length of the buffer is used.

The **read** and **write** forms of the **perm** command are for use in saving and restoring a permutation list on the disk between learner sessions.

The structure of the buffer is as follows:

Item	Bytes
Version of **perm**	2
Length of list	2
Number of remaining items	2
Item bitmap	N

The item bitmap is formatted such that each bit corresponds to a number in the permutation list: if the bit is "1", the item is still available; 0 indicates it has been removed. For example, if bit 3 is set to 1, the number 3 has not yet been removed from the permutation list. The length of the item bitmap is determined by the length of the list. Since each byte can hold the status of 8 items, N is the length of the list divided by 8, rounded up. It can be computed as int((length+7)/8) or ((length+7) ÷ 8).

perm write; permBufter [,length] [; 'NAME' | default]

Writes a given length of bytes from a buffer into a permutation list. If the length is not specified, the defined length of the buffer is used.

System Variable

zreturn

- 2	Redundant operation: replace used with number already in list or remove used with number already removed
- 1	Operation Successful
3	Out of range: attempt to create list larger than 32,767 or attempt to remove or replace number larger than list size
6	Duplicate name using create
10	Name not found
17	Attempt to write invalid permutation list data
18	Insufficient space in memory pool

BNSDOCID: <WO___9907007A2_I_>

polygon

Draws a polygon

polygon [LOCATION]; [LOCATION]; [LOCATION] ... [;fill]

Description

Draws a polygon in the foreground color using the specified LOCATION as vertices. The optional modifier fill specifies that the polygon be filled. Only convex polygons can be filled; concave ones must be split into multiple convex ones (see below). A minimum of three points must be specified. A maximum of 50 points may be specified. If the last point does not match the first, they are connected.

A polygon is affected by scale, rotate and width. Filled polygons function only when any horizontal line intersects exactly two points on the polygon.

Splitting a concave polygon into multiple convex polygons will produce the desired shape

```
polygon  0,100; 200,0; 200,100            $$ draws a triangle
polygon  10:10; 10:20; 20:20; 20:10 $$ draws a rectangle
polygon  0,100; 200,0; 200,100; fill      $$ draws filled polygon
```

press

Generates an input value.

press *KEY* | expression [;**first**]

press *KEY* | expression [,inputF [,inputX,inputY [,inputA]]] [; **first**]

Description

The **press** command puts a specified key into the input buffer as if pressed on the keyboard. At the next input-processing point (**pause**, **arrow**, **nextkey**, or end-of-unit), the key is

processed and performs whatever function a similar real keypress does including the update of the system variable **zinput**

The optional modifier **first** causes the key to be inserted before any other keys waiting in the input buffer: it will be processed first.

Optionally, the values for the system variables **zinputf, zinputx, zinputy,** and **zinputa** can be specified so as to be set along with **zinput** when the key is processed. If a value is not specified, the corresponding system variable is set to zero.

The *KEY* values for **press** are limited: embedding and the simple a,b,c form are not available. However, the key can be an expression such as a constant, variable or calculation.

Examples

Example 1

Puts %space into the input buffer as if it had been pressed on the keyboard.

```
press    %space
```

Example 2

Places %F1 in the input buffer before any other keys waiting to be processed; it will be processed first.

```
press    %F1; first
```

Example 3

Simulates a pointer click by placing the key value %pointer along with location and area id information into the input buffer. At the next input-processing point, **zinput=**%pointer, **zinputx=**320, **zinputy=**240, and **zinputa=5.** Since no value is specified for **zinputf,** it is set to 0.

```
press    %pointer,,320,240,5
```

Example 4

After an unsuccessful try at the question, the learner is given the first few letters of the correct answer typed in as input to the arrow.

```
write    Who is buried in Grant's tomb?
arrow
answer   «U,Ulysses,S» Grant
no
.        write    Press any key for a hint.
.        pause
.        press    %enter          $$ erases the mistake
.        press    %"G"
.        press    %"r"
endarrow
```

System Variable

zreturn

-1	Operation successful
0	Key buffer overflow

print

Sends output to the printer, or changes print parameters.

print	variable, length
print	**initial**; **prn** | **lpt1** | **lpt2** | **lpt3** | **com1** | **com2**
print	**initial** ; **file**, *'FILENAME'* [,length]
print	**end**
	buffer variable buffer to output to printer
	length number of bytes to send to the printer
	initial changes the printer device
	end ends network spooling and resets printer to **lpt1**

Description

Sends the specified number of bytes to the standard printer device **lpt1**, or to another device or file specified by **print initial.**

Legal devices for **print** are:

prn, lpt1, lpt2, lpt3, com1, and com2.

The **initial** keyword changes the device used for printing. With the **file** form, a file name is specified. The ASCII filename string may include a drive and path, and must be terminated with a null byte. If the length is not specified, the defined length of the variable is used.

If the specified file already exists, it is initialized to 0 length in preparation for printing.

The **end** keyword clears any printer connection established with **print initial**. On a network system, it also terminates printer output spooling and initiates printing.

Unless otherwise specified by **print initial**, output is sent to **lpt1**.

print only sends the specified characters to the printer. Printer control characters are not added to the output. Refer to the manufacturer's manual for printer control codes.

Most printers require an ASCII carriage return code (h0d) or a carriage return/line feed sequence (h0d0a) at the end of each line to position the print head and paper for the next line.

Technical Note: When no printer connection is in effect, printer output is sent to the BIOS INT 17H printer driver, device 0 which is normally **lpt1** unless it has been redirected. When a printer connection is active, printer output is sent to the device via the standard DOS "write" call.

Examples

Example 1

Prints the title line for a report. The user's name is stored in *usemame.* Note that a carriage return/line feed sequence is included.

9NSDOCID: <WO 9907007A2 I >

```
define   local
username, 20            $$ name of user
text, 100               $$ text to print
texten, 2               $$ length of text to print
CRLF = h0d0a            $$ carriage return and line feed
define   end
*
packz    text;textlen;Test results for «showa,username»
         « showa,CRLF »
*
print    text, textlen  $$ print the line
```

Example 2

Redirects future printing to **lpt2**.

```
print    initial; lpt2  $$ initialize lpt2
```

Example 3

Redirects future printing to file *C:\TENPRINT.PRN*

```
define   local
filename,   40
define   end
packz    filename;;c:\tenprint.prn  $$ put filename in variable
print    initial; file, filename    $$ redirect output to file
```

System Variable

zreturn

- 1	Operation Successful
0	Disk error (see **zdoserr** and **zedoserr**)

put

Substitutes one string of characters for another.

put	*fromTEXT; to TEXT* [,buffer [,length]]	
	fromTEXT	text string to find and replace
	toTEXT	replacement text sting (can be null)
	buffer	user variable buffer on which to perform replacement
	length	maximum length for buffer

Description

Replaces every occurrence *of fromTEXT* with *toTEXT* in the student response buffer at an arrow or optionally in any specified buffer. The *TEXT* occurs without quotes and can contain embedded show commands.

Examples

Example 1

Shows how put can be used to change an abbreviation to its full word equvalence in answer judging.

```
write     What is the biggest US city?
arrow     10:10
put       NY, New York
answer    New York
.         write    Great!
endarrow
```

Example 2

Replaces all occurances of "dog" with "elephant" in the buffer *mystring.*

```
put       dog,elephant,mystring
```

BNSDOCID: <WO 9907007A2 I >

Example 3

Embedded text can be used for both the *from* and *to* strings.

```
put      «a,oldchars,oldlen»,«a,newchars,newlen»,textvar,length
```

Example 4

This produces a 24-hour type clock display. If the hours or minutes are less than 10 the tens digit would contain a space from the **showt**'s of the **pack.** The **put** replaces any spaces in *var* with the digit 0.

```
pack     var, length,«showt, hrs, 2»:«showt, mins, 2»
put       ,0, var, length      $$ from string is just a space
*                              replace spaces with 0
showa    var, length
```

System Variables

zreturn

-1	Operation successful, all possible replacements made
0	Insufficient room for replacement

Miscellaneous

zlength	Length of the text string after replacement
znumber	Number of replacements made
zcount	Length of response buffer, if used

receive

Receives argument values passed to a subroutine

---

receive argument/16s

Description

Receives arguments passed to a subroutine unit and places their values in the specified variables. Null arguments are specified by blank entries (,,)in the argument list.

Arguments can be passed by **do, library, jump, and jumpop** commands. The following applies to **receive**:

- Any element in the argument list may be null. A null element in the argument list causes the corresponding passed value to be ignored.
- A litteral used as a receive argument causes a condense error.
- Only numeric integer and real variables are valid in the argument list. A text variable of eight characters or less is considered a numeric in this context.
- Any number of **receive** commands may be executed, and any **receive** may be executed an arbitrary number of times. Each time a **receive** is executed, the originally passed arguments are used.
- Intervening calls to other units (which can receive or return their own arguments) have no effect on the values received in the current unit (passed values are local to the individual units).
- If a null argument is sent, the corresponding element in the **receive** list is not changed.
- Normally, the number of arguments received must match the number passed, otherwise an execution error occurs. This can be overridden by using **nocheck receive.**

Example

Unit *one* passes arguments to subroutine *circles.*

SNSDOCID: <WO 9907007A2 >

one

```
.
.
.
do      circles(160, 100, 5)
.
.
.
```

circles

```
define  local
x,      2  $$ center of circles
y,      2
n,      2  $$ number of circles
r,      2  $$ radius of circle
define  end
*
receive x, y, n
at      x,y
loop    r <= 10, n * 10, 10
.       circle  r
endloop
```

System Variables

zargsin	Number of arguments (including nulls) sent to subroutine
zargsout	Number of return arguments expected
zargs	Number of non-null arguments actually received

reloop

Returns to the beginning of a lOOP structure.

---

reloop [*CONDITION*]

Description

Used to restart the loop structure. If the tag is blank or the condition is true, **reloop** returns to the beginning of the loop structure as if **endloop** had been executed. If the condition is false, execution continues with the next command. **reloop** always appears at the same indentation level as the **loop** command to which it returns, regardless of any other indented structures surrounding the **reloop**.

For more information on loop structures, see the loop command.

Example

An *ITERATIVE* loop which skips the case when the *counter* is 13.

```
define   local
index, 2    $$ loop counter
define   end
*
loop     index <= 1,20
reloop   index = 13
.        at index:5
.        write    This is line «show, index».
endloop
```

return

Ends subroutine execution optionally returning arguments.

---

return [argument/16s]

Description

Exits immediately from the current unit if it was called by **do** or **library**. If the unit was not called by a **do** or **library**, an execution error occurs.

The blank-tag form causes an exit from the unit with no arguments passed back to the calling unit.

If a list of return arguments is specified, the values are returned to the calling unit. A null entry in the return list causes the corresponding value in the receive argument list to remain unchanged.

See the **receive** and **nocheck** command descriptions for more information on argument passing.

Example

Unit *test* calls subroutine *square*, passing it one argument; subroutine *square* receives that argument, uses it in a calculation, then returns the resulting value to the variable *numsquar* back in unit *test*

```
test
.
.
do        square(12;numsquar)
show      numsquar
.
```

```
square
define   local
number
number2
define   end
*
receive  number
calc     number2 <= number * number
return   number2
```

System Variables

Miscellaneous

zargsin	Number of arguments, including nulls, sent to subroutine
zargsout	Number of arguments the calling **do** or **library** expects returned
zargs	Number of non-null arguments actually received by **receive**

rotate

Rotates subsequent graphic objects.

---

rotate [*degrees* [;*LOCATION*]]

Description

Causes all subsequently plotted graphic objects to be rotated by the specified number of *degrees*. Rotation occurs counter-clockwise about the plotting origin, as specified by the **origin** command, unless an optional location is specified. Specifying a rotational origin affects only subsequestly rotated objects.

The system default is **rotate 0** (no rotation), set at each main unit and by **initial** and **status restore; default.** The blank-tag form also sets the standard default of no rotation. The default can be changed using **rotate** followed by **status save; default**.

The effect of the graphic tranformation commands, including **rotate**, can be temporarily suspended with **enable absolute** and restored **with disable absolute.**

The display control commands are applied in the following order:

scale

rotate

aspect ratio correction

origin

window relative coordinates

window clipping

The following objects are not subject to rotation:

text

images

windows

boxes and erases specified using characters and lines (e.g., **box 10,3)**

To rotate text, see the **text rotate** command.

Example

A box is continuously rotated at the center of the screen.

```
define   local
degree   , 2
define   end
*
```

```
origin    zdispx/2, zdispy/2          $$ middle of the screen
mode      xor
loop
.         rotate    degree
          box       -50,-50; 50,50;1  $$ plot box
          delay     01
.         box       -50,-50; 50,50;1  $$ erase box
.         calc       degree <= degree +1
endloop
```

System Variables

zrotate Number of degrees of rotation.

zrotatex The X coordinate of the rotation origin.

zrotatey The Y coordinate of the rotation origin.

scale

Scales graphic objects.

---

scale [factor | Xfactor, Yfactor [*LOCATION*]]

Description

Causes all subsequently plotted graphic objects to be scaled by the given factor or separate x (horizontal) and y (vertical) factors may be specified. For example, **scale .5** would make all graphics objects 1/2 size, while **scale 2,3** would double graphics in the x direction, and triple them in the y direction. Scaling occurs relative to the plotting origin, as specified by the **origin** command, unless an optional scaling origin location is specified.

Specifying a scaling origin affects only subsequently scaled objects.

BNSDOCID: <WO 9907007A2 1 >

The standard default is **scale 1** (no scaling), set at each main unit and by **initial** and **status restore; default.** The blank-tag form also sets the standard default of no scaling. This default can be changed with **status save; default.**

The effect of the graphic transform commands, including **scale,** can be temporarily suspended with **enable absolute** and restored with the **disable absolute**.

The display graphic transform are applied in the following order:

scale

rotate

aspect ratio correction

origin

window relative coordinates

window clipping

The following objects are not subject to scaling:

text

images

windows

boxes and erases specified using characters and lines (e.g., **box 10,3)**

Example

Produces a series of concentric boxes.

```
define   local
factor,  2
define   end
*
origin   zdispx/2, zdispy/2
*
```

```
loop    . factor <= 1,10
.         scale    factor
.         box      -10,-10;10,10;1
endloop
```

System Variables

zscalex	The horizontal scaling factor (8-byte real)
zscaley	The vertical scaling factor (8-byte real)
zscaleox	x-coordinate of the scaling origin
zscaleoy	y-coordinate of the scaling origin

screen

Sets or tests screen resolution type.

screen	[**test,**] *type* [,*mode*] [,*resolution*] [,*chrome*]
screen	[**test,**] **variable,** type, mode, resolution, chrome
screen	**native** | **edit**
	type keywords and values:
	cga 0
	ega 3
	vga 10
	mcga 11
	evga 1
mode	keywords and values
	graphics 0
	text 1
resolution	keywords and values
	low 0
	medium 1
	high 2
	alt1 3
	alt2 4
	alt3 5
chrome	keywords and values:
	color 0
	mono 1
test:	Only check if specified settings work and return result in **zreturn,** don't actually change settings.
ative:	Set screen to highest possible settings (up to **vga**) for given display driver.
edit:	Put settings to editor's settings.
variable: Use numeric values instead of keywords for type, mode, resolution, and chrome.	

Description

Sets the screen hardware to the specified type, mode, resolution, and chrome. The choice of a screen setting is limited by the display driver, display controller and the monitor being used. In producing courseware that is to run on a wide variety of IBM PCs directly using the DOS software platform, one needs to select a screen setting that is compatible with the lowest common display hardware.

Normally, the **screen** command occurs once at the beginning of a lesson as in the **+initial** control block.

When a **screen** command changes screen settings, the screen is erased, the display page is reset to 1, and all plotting parameters are reset to **standard** as if the following statements are executed:

```
status   restore; standard
status   save; default
```

Shorthand Notation

The most popular screen settings can be selected without inputting all arguments. Graphics and color are the default settings. The resolution usually defaults to the standard value for the display controller type.

Shorthand	Fully Specified Form
screen cga	cga,graphics,medium,color
screen, cga,high,mono	cga,graphics,high,mono
screen, ega	ega,graphics,high,color
screen ega,medium	ega,graphics,medium,color
screen ega,low	ega,graphics,low,color
screen vga	vga,graphics,medium,color
screen mcga	mcga,graphics,medium,color
screen mcga,high,mono	mcga,graphics,high,mono
screen evga	evga,graphics,high,color
screen evga,medium	evga,graphics.medium.color

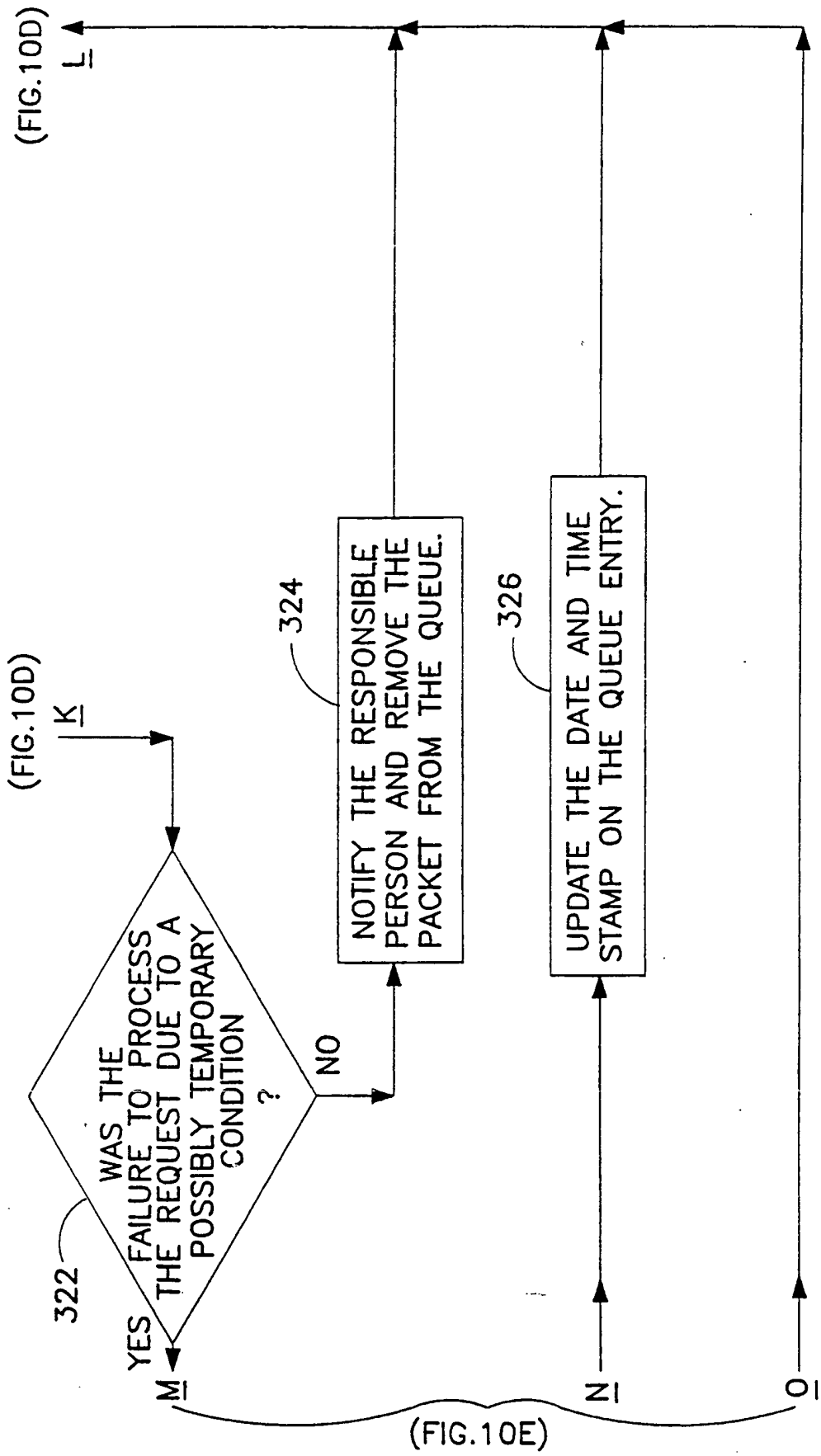


FIG.10F

ENSDOCID <WO 9907007A2 I >

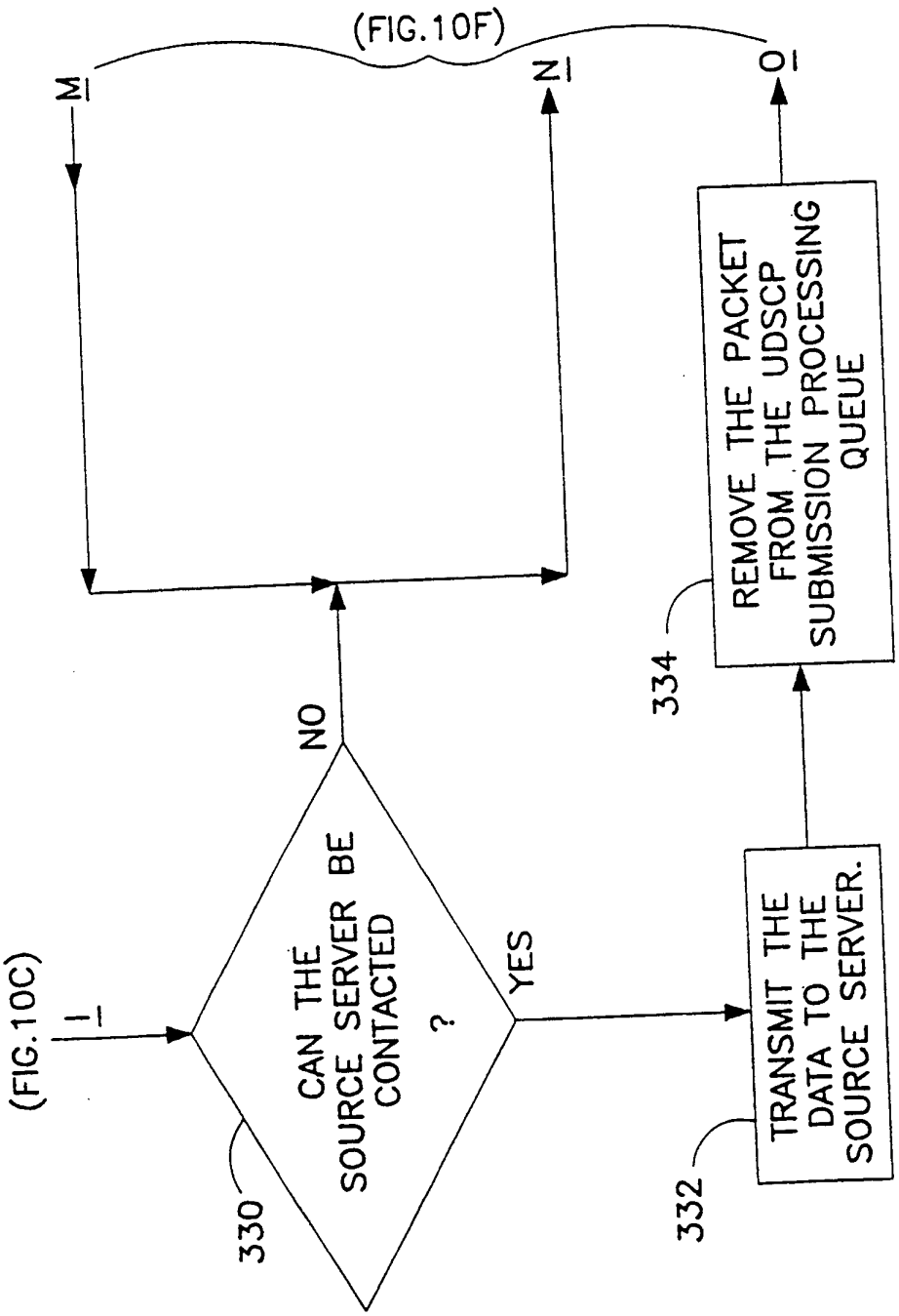


FIG.10E

BNSDOCID <WO 9907007A2 I >

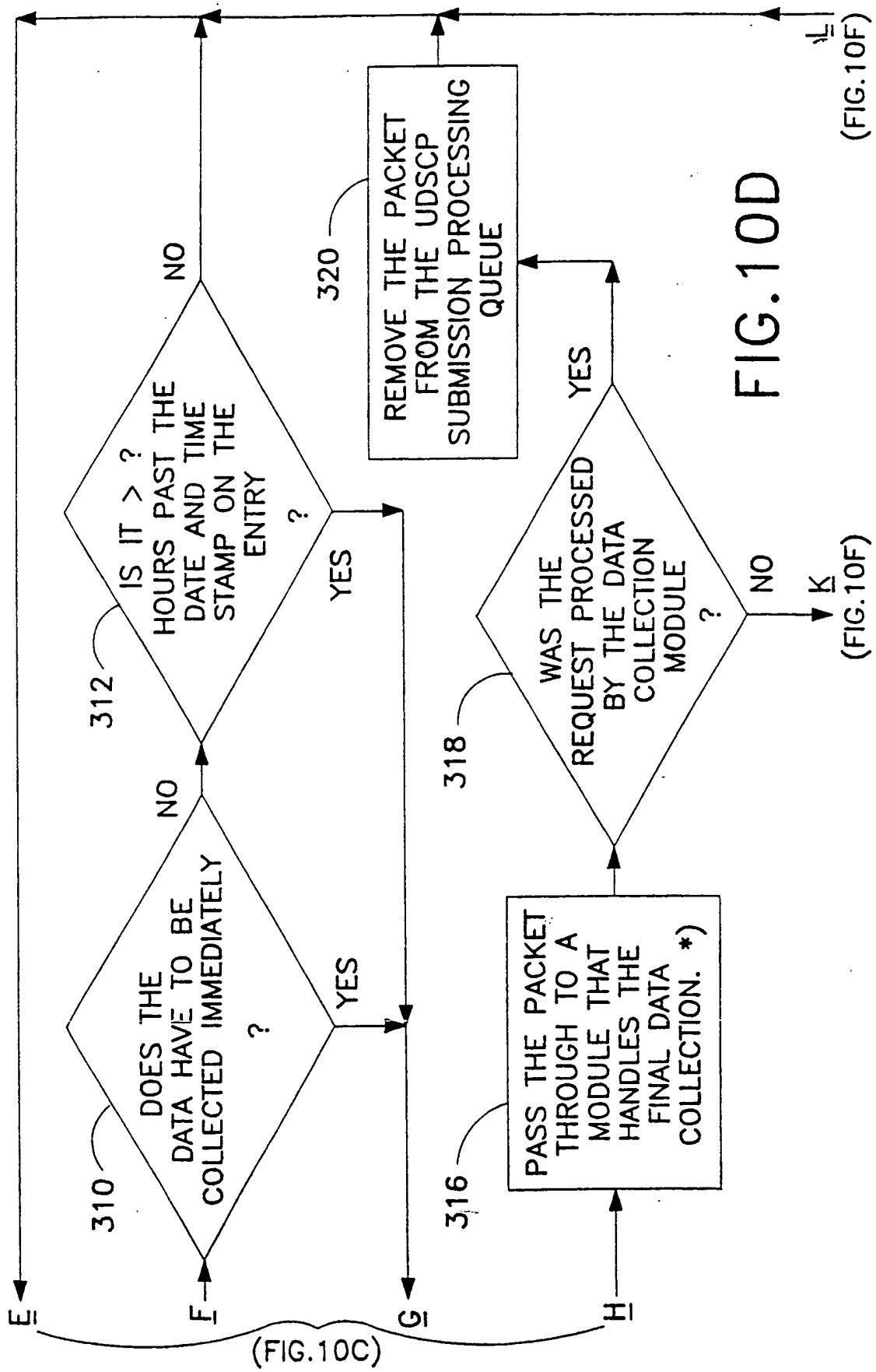


FIG.10D

3NSDOCID: <WO 9907007A2 I >

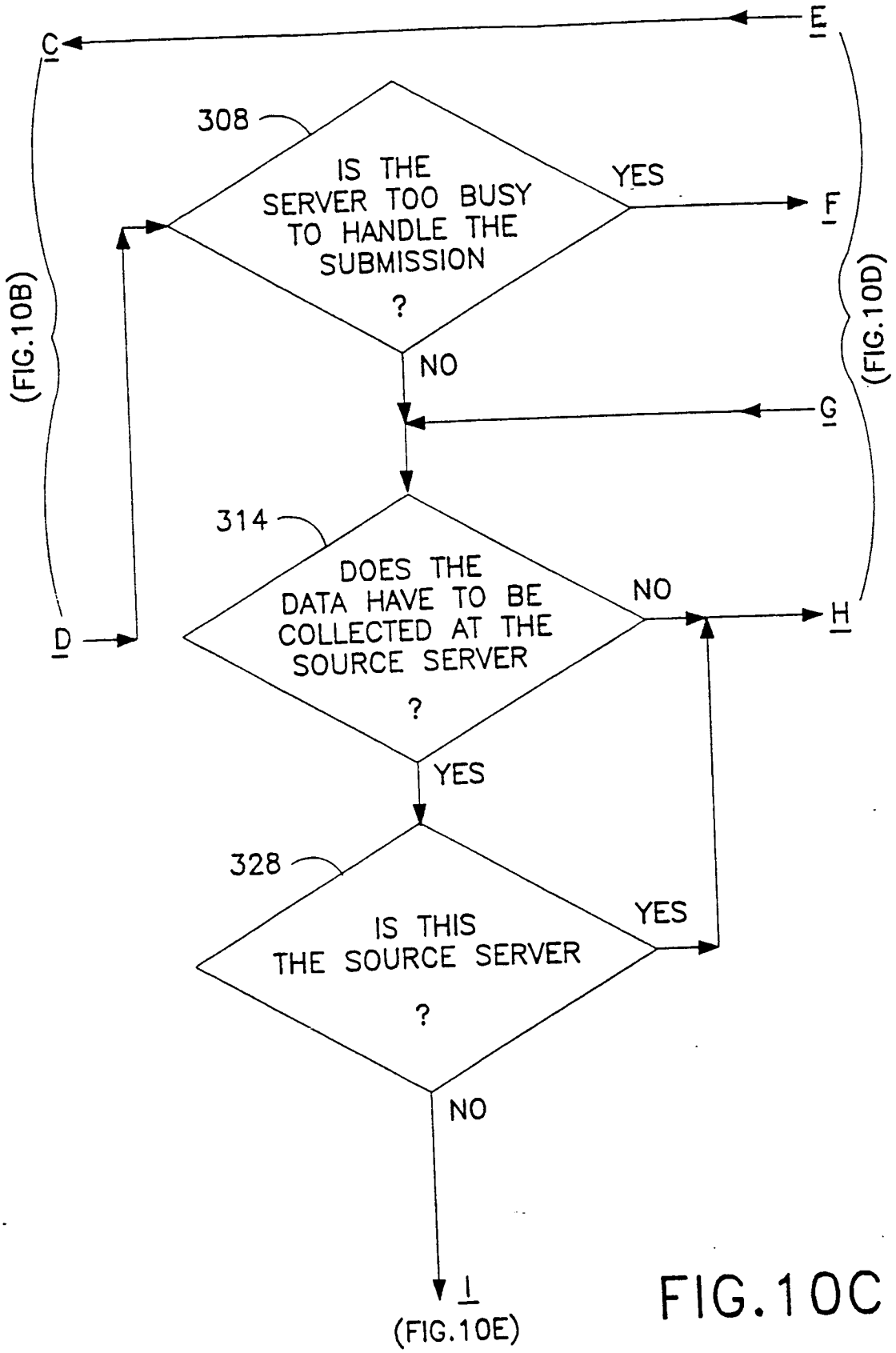

E
C
308
IS THE SERVER TOO BUSY TO HANDLE THE SUBMISSION ?
YES
F
(FIG.10B)
(FIG.10D)
NO
G
314
DOES THE DATA HAVE TO BE COLLECTED AT THE SOURCE SERVER ?
NO
H
D
YES
328
IS THIS THE SOURCE SERVER ?
YES
NO
I
(FIG.10E)

FIG.10C

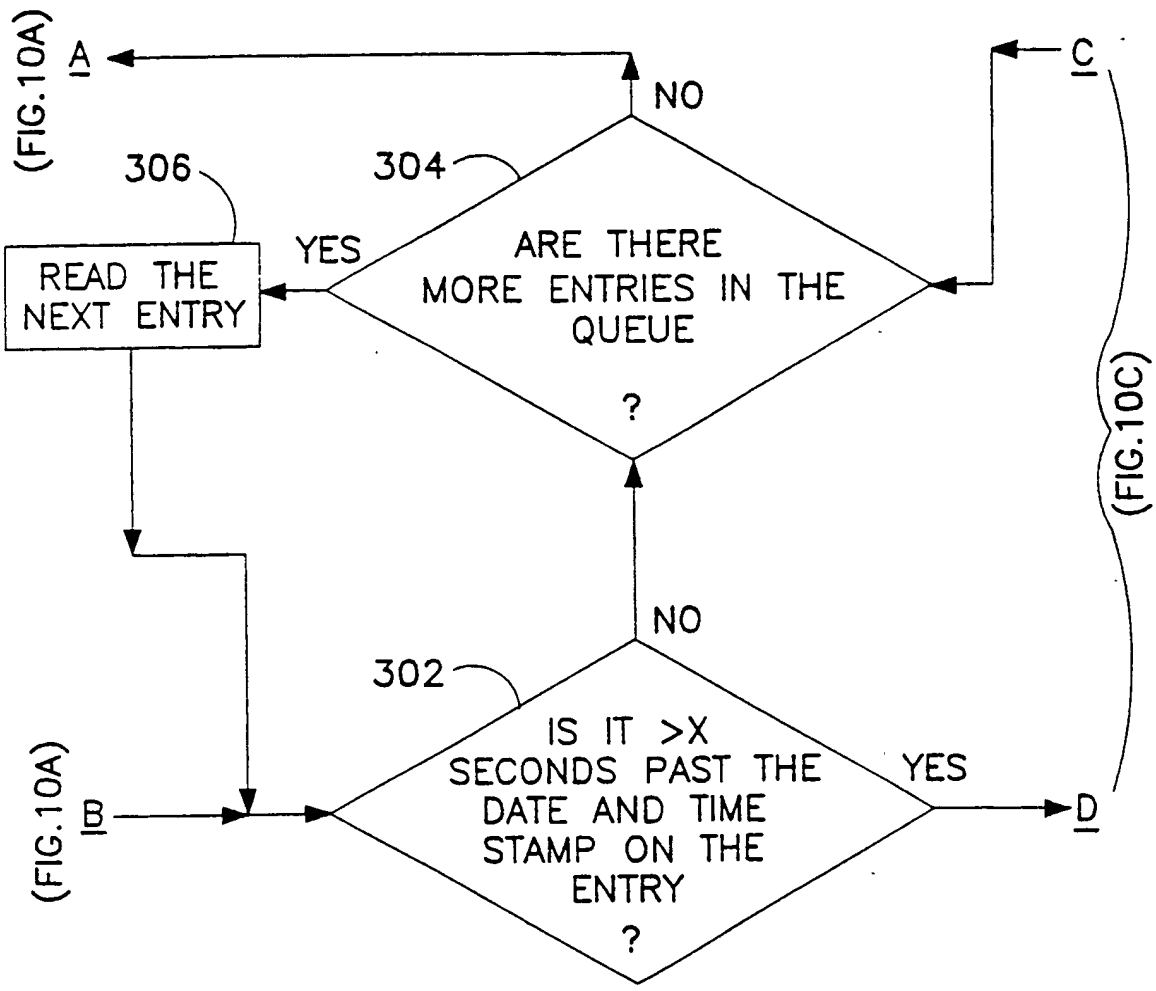


FIG.10B

BNSDOCID <WO___9907007A2_I_>

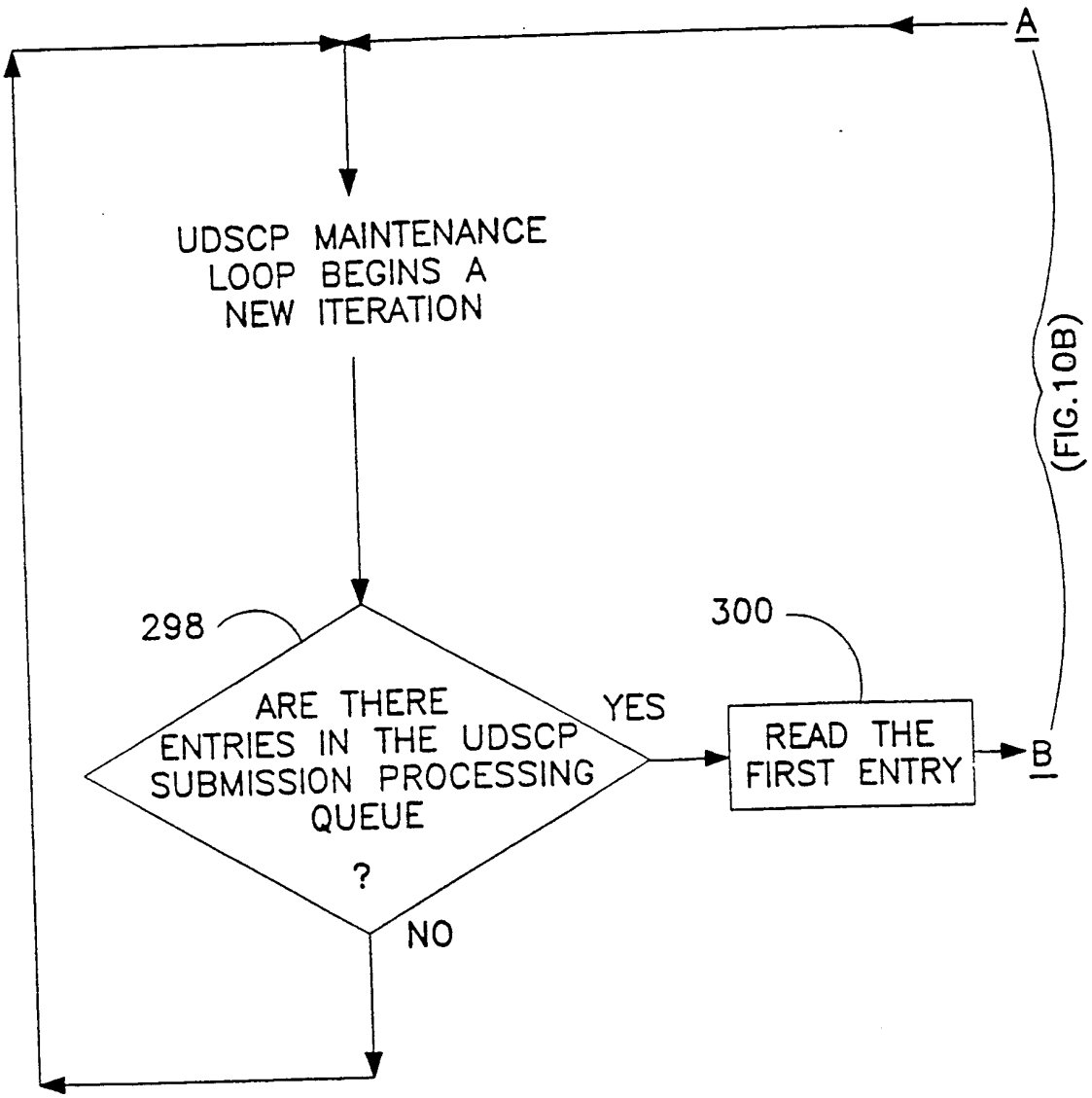
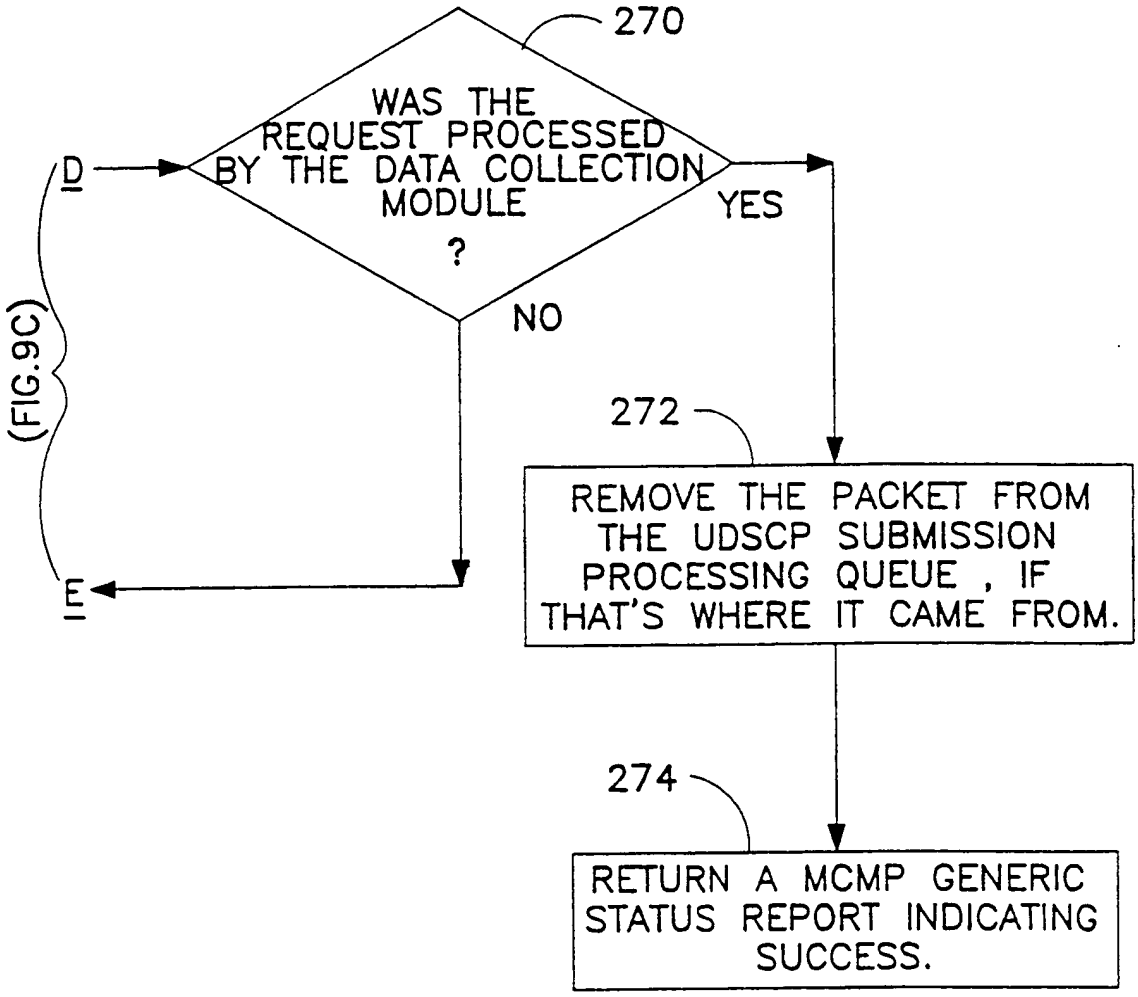

A
UDSCP MAINTENANCE LOOP BEGINS A NEW ITERATION
298
ARE THERE ENTRIES IN THE UDSCP SUBMISSION PROCESSING QUEUE ?
YES
NO
300
READ THE FIRST ENTRY
B
(FIG.10B)

FIG.10A



*) THIS IS THE MODULE THAT HANDLES E-MAILING THE SUBMISSION, RECORDING IT IN A DATABASE, WRITING IT TO A FILE, OR PERFORMING ANY OTHER NECESSARY SERVER-SIDE TASK WITH THE DATA..

FIG.9D

BNSDOCID: <WO 9907007A2 I >

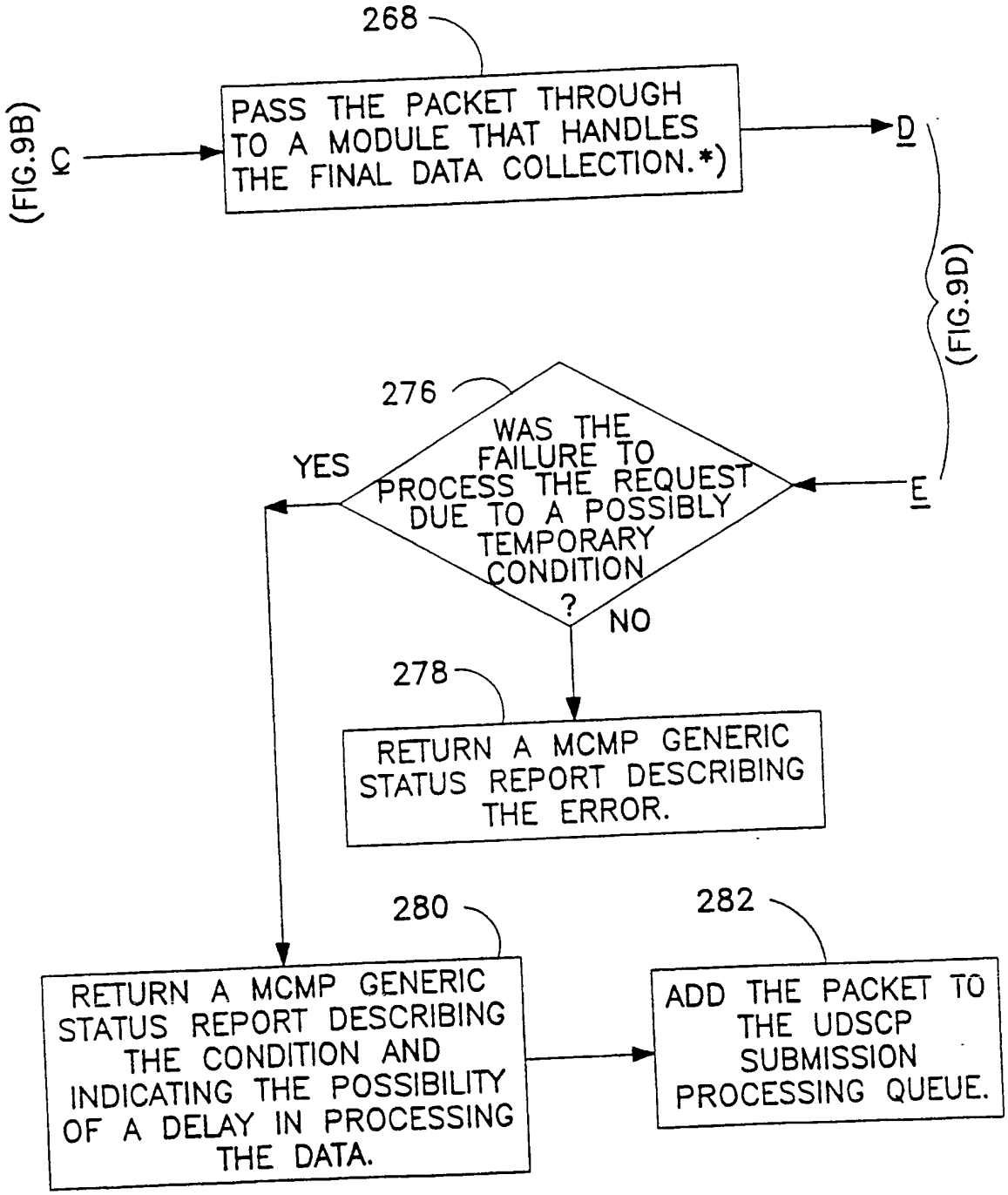

268
PASS THE PACKET THROUGH TO A MODULE THAT HANDLES THE FINAL DATA COLLECTION.*)
(FIG.9B) C
D
(FIG.9D)
276
WAS THE FAILURE TO PROCESS THE REQUEST DUE TO A POSSIBLY TEMPORARY CONDITION ?
YES
NO
E
278
RETURN A MCMP GENERIC STATUS REPORT DESCRIBING THE ERROR.
280
RETURN A MCMP GENERIC STATUS REPORT DESCRIBING THE CONDITION AND INDICATING THE POSSIBILITY OF A DELAY IN PROCESSING THE DATA.
282
ADD THE PACKET TO THE UDSCP SUBMISSION PROCESSING QUEUE.

FIG.9C

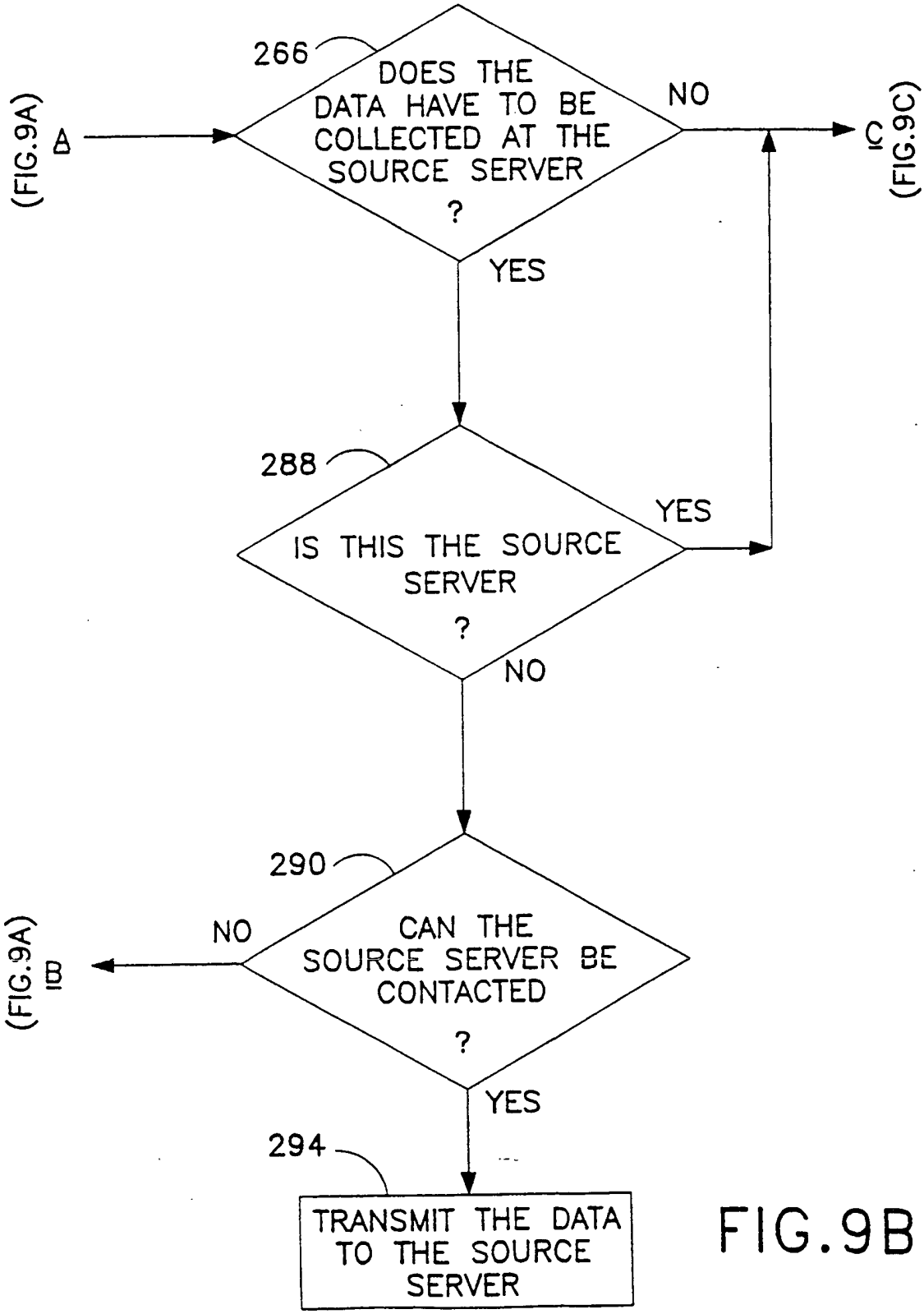

(FIG.9A) A
266
DOES THE DATA HAVE TO BE COLLECTED AT THE SOURCE SERVER ?
NO
C (FIG.9C)
YES
288
IS THIS THE SOURCE SERVER ?
YES
NO
290
CAN THE SOURCE SERVER BE CONTACTED ?
NO
(FIG.9A) B
YES
294
TRANSMIT THE DATA TO THE SOURCE SERVER

FIG.9B

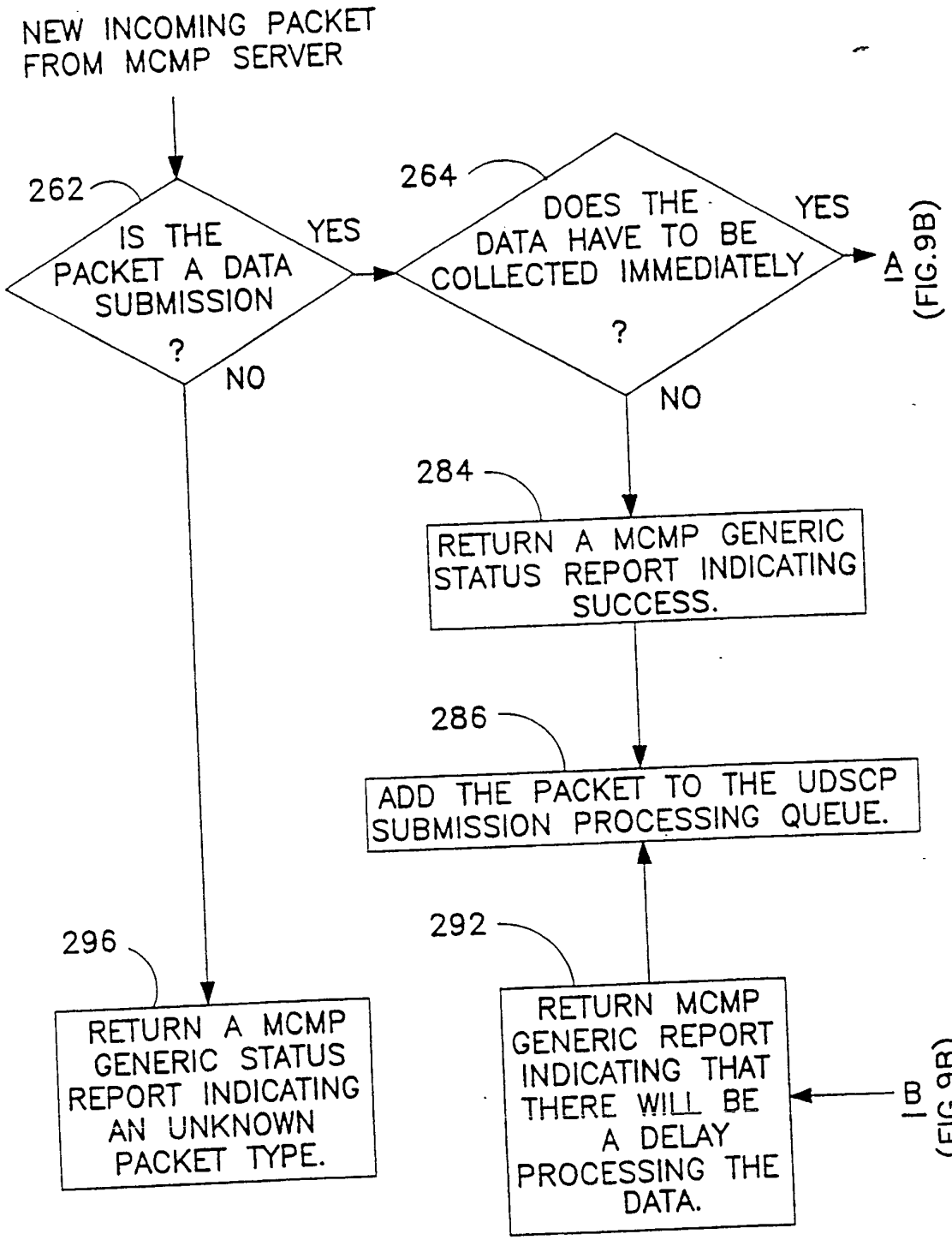

NEW INCOMING PACKET FROM MCMP SERVER
262
IS THE PACKET A DATA SUBMISSION ?
YES
NO
264
DOES THE DATA HAVE TO BE COLLECTED IMMEDIATELY ?
YES
NO
A
(FIG.9B)
284
RETURN A MCMP GENERIC STATUS REPORT INDICATING SUCCESS.
286
ADD THE PACKET TO THE UDSCP SUBMISSION PROCESSING QUEUE.
292
RETURN MCMP GENERIC REPORT INDICATING THAT THERE WILL BE A DELAY PROCESSING THE DATA.
B
(FIG.9B)
296
RETURN A MCMP GENERIC STATUS REPORT INDICATING AN UNKNOWN PACKET TYPE.

FIG.9A

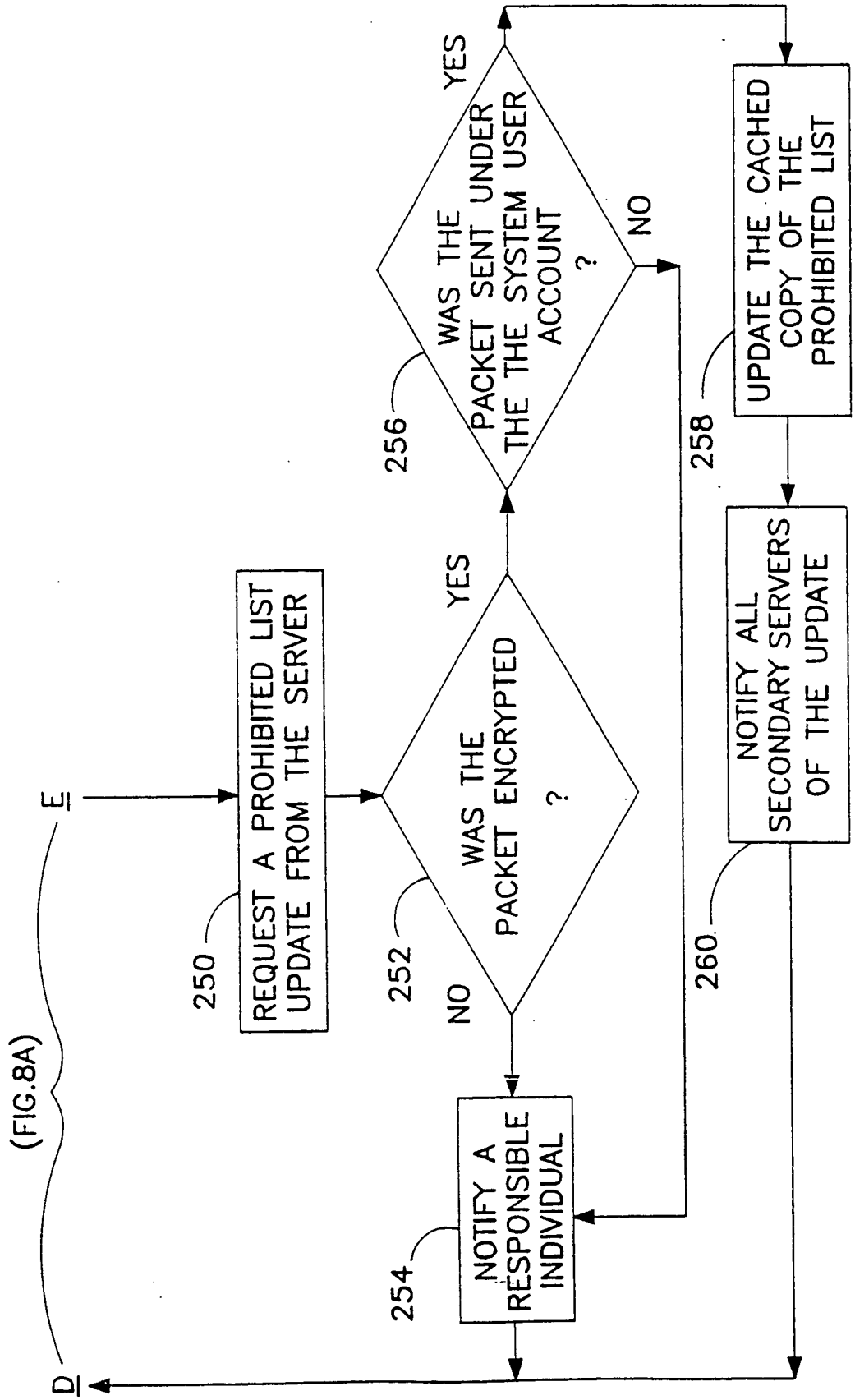


FIG.8C

ENSDOCID <WO___9907007A2_I_>

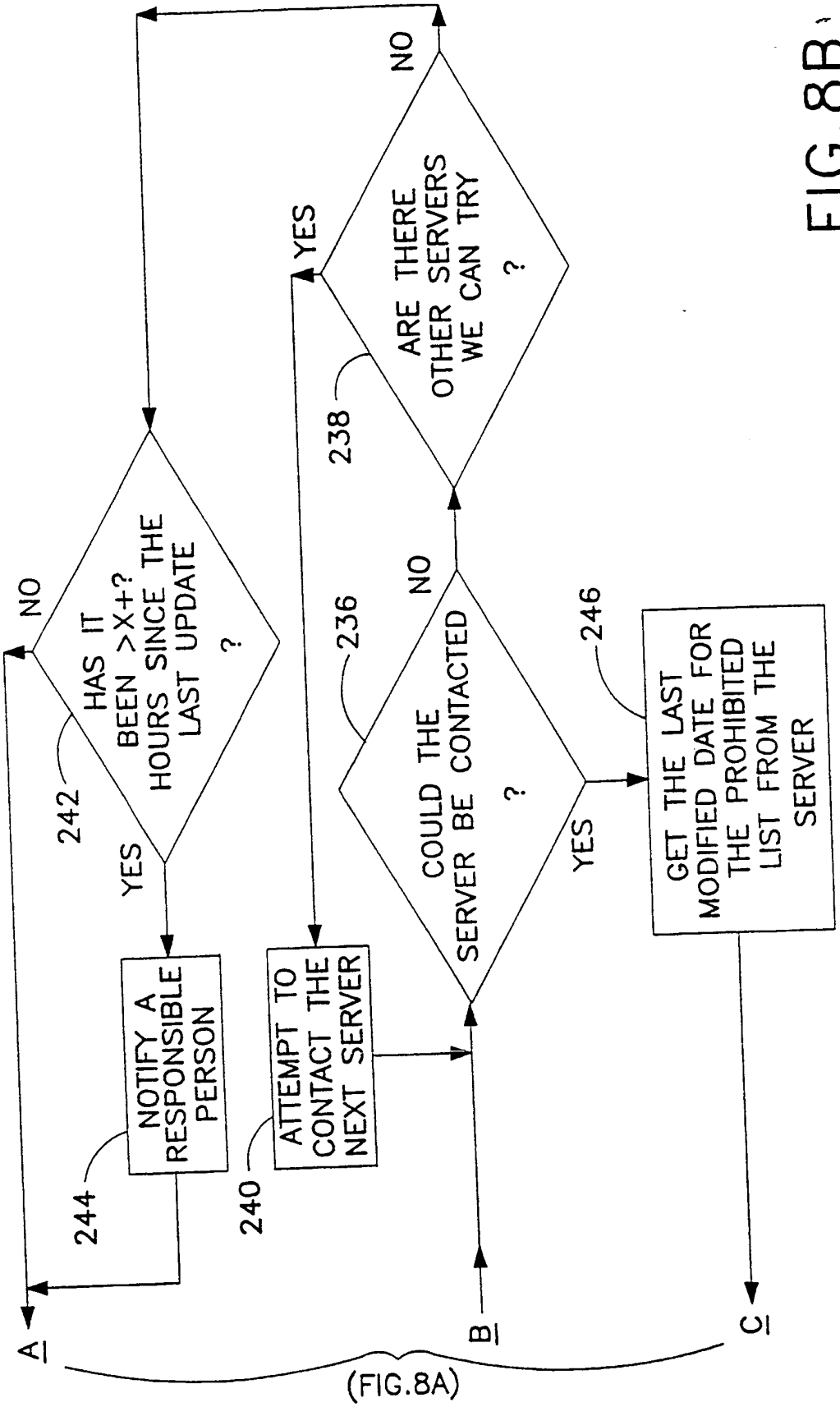


FIG.8B

(FIG.8A)

BNSDOCID <WO 9907007A2 I >

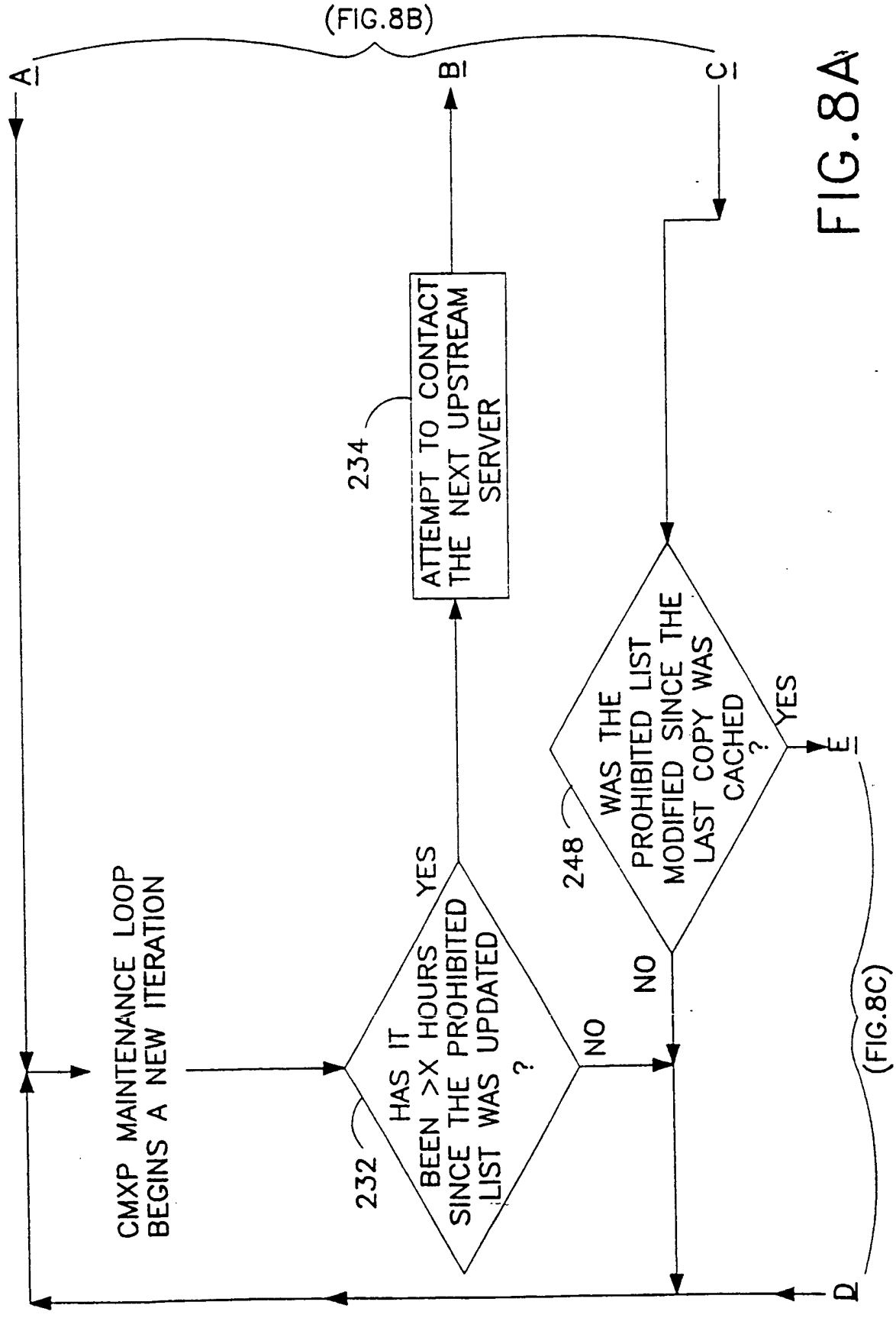


FIG.8A

RNSDOCID: <WO 9907007A2 I >

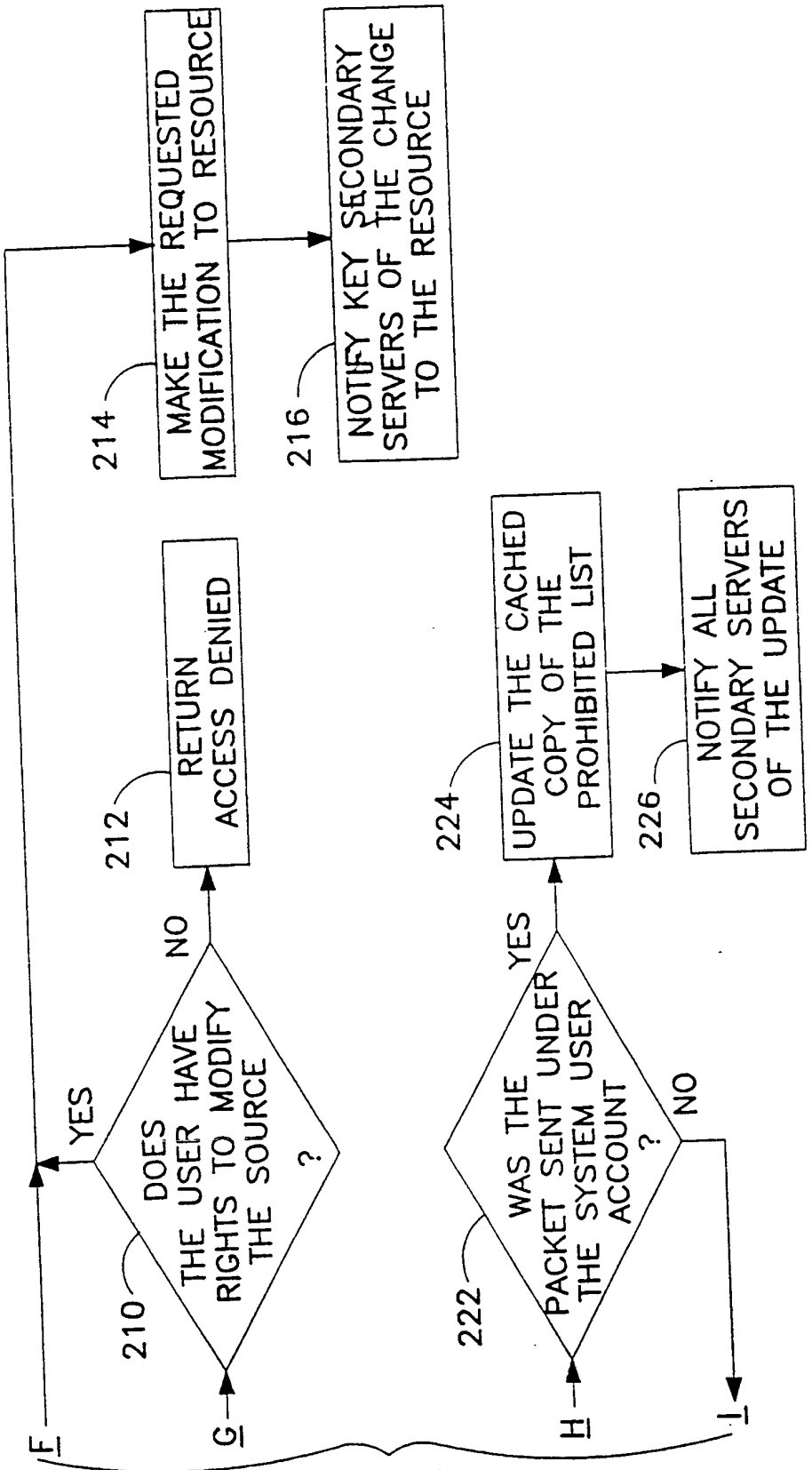


FIG. 7E

(FIG.7D)

BNSDOCID <WO 9907007A2 I >

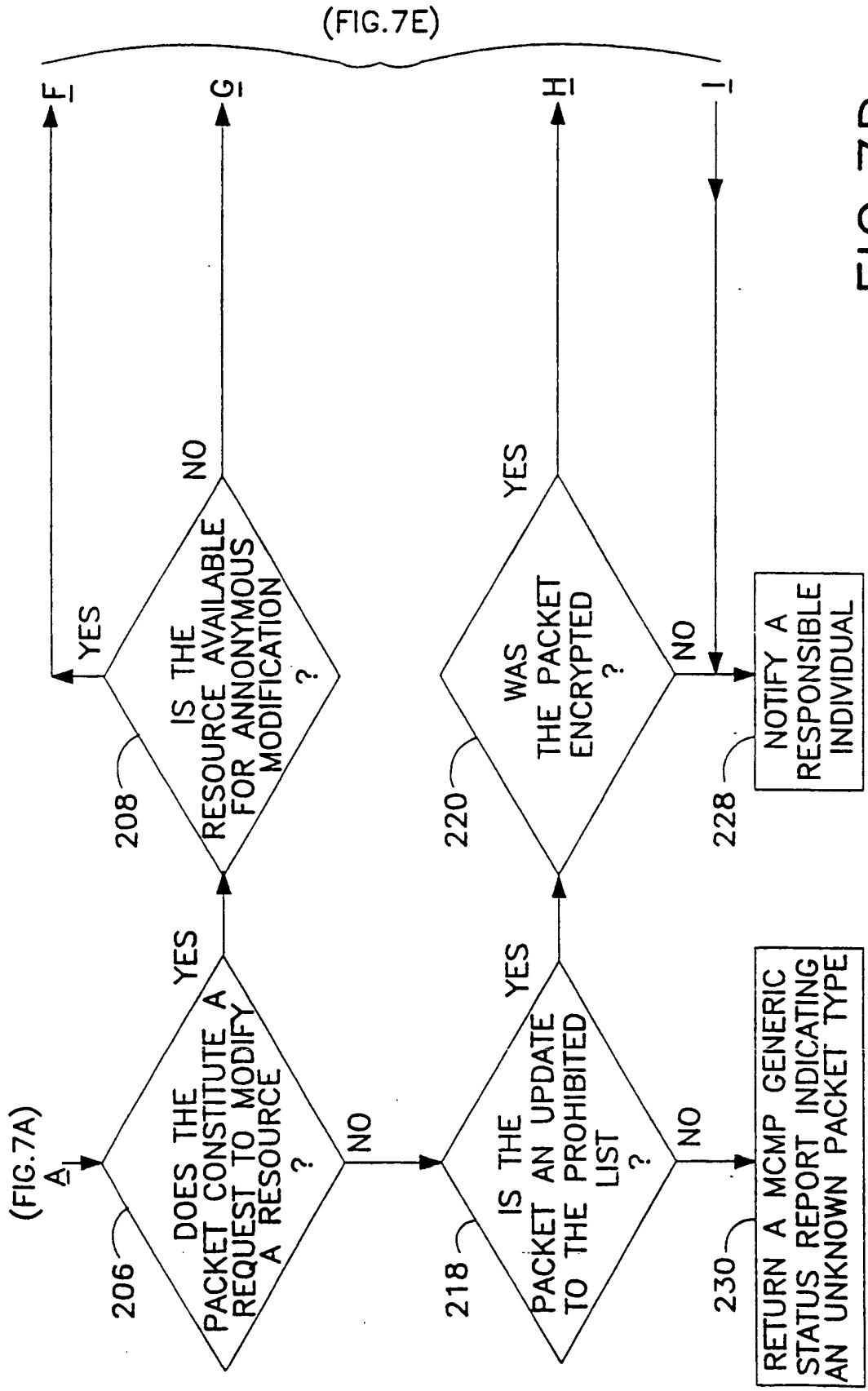
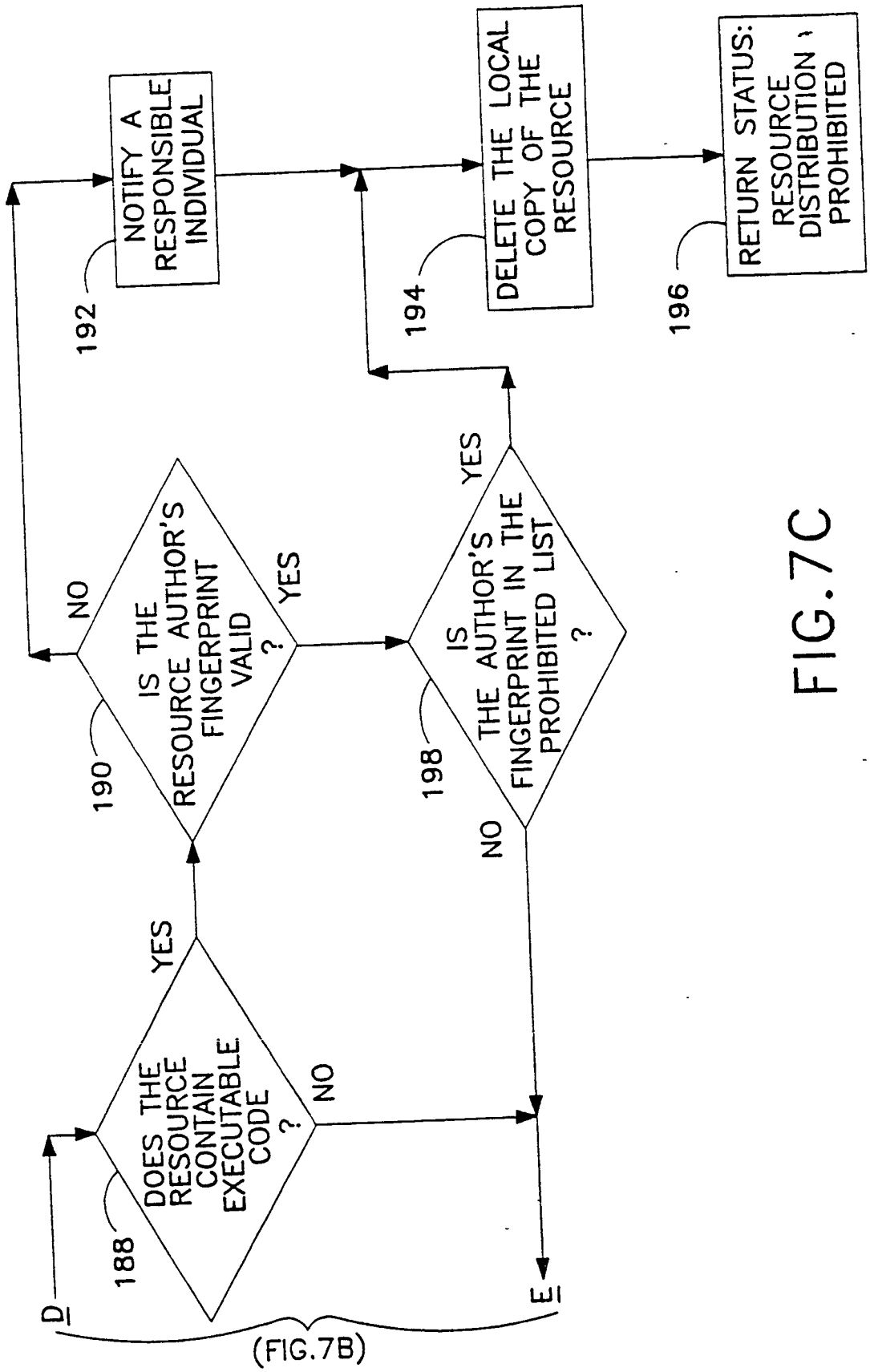


FIG.7D

ENSDOCID: <WO_ _9907007A2_I_>



(FIG.7B)

FIG.7C

BNSDOCID <WO 9907007A2 I >

FIG.7C

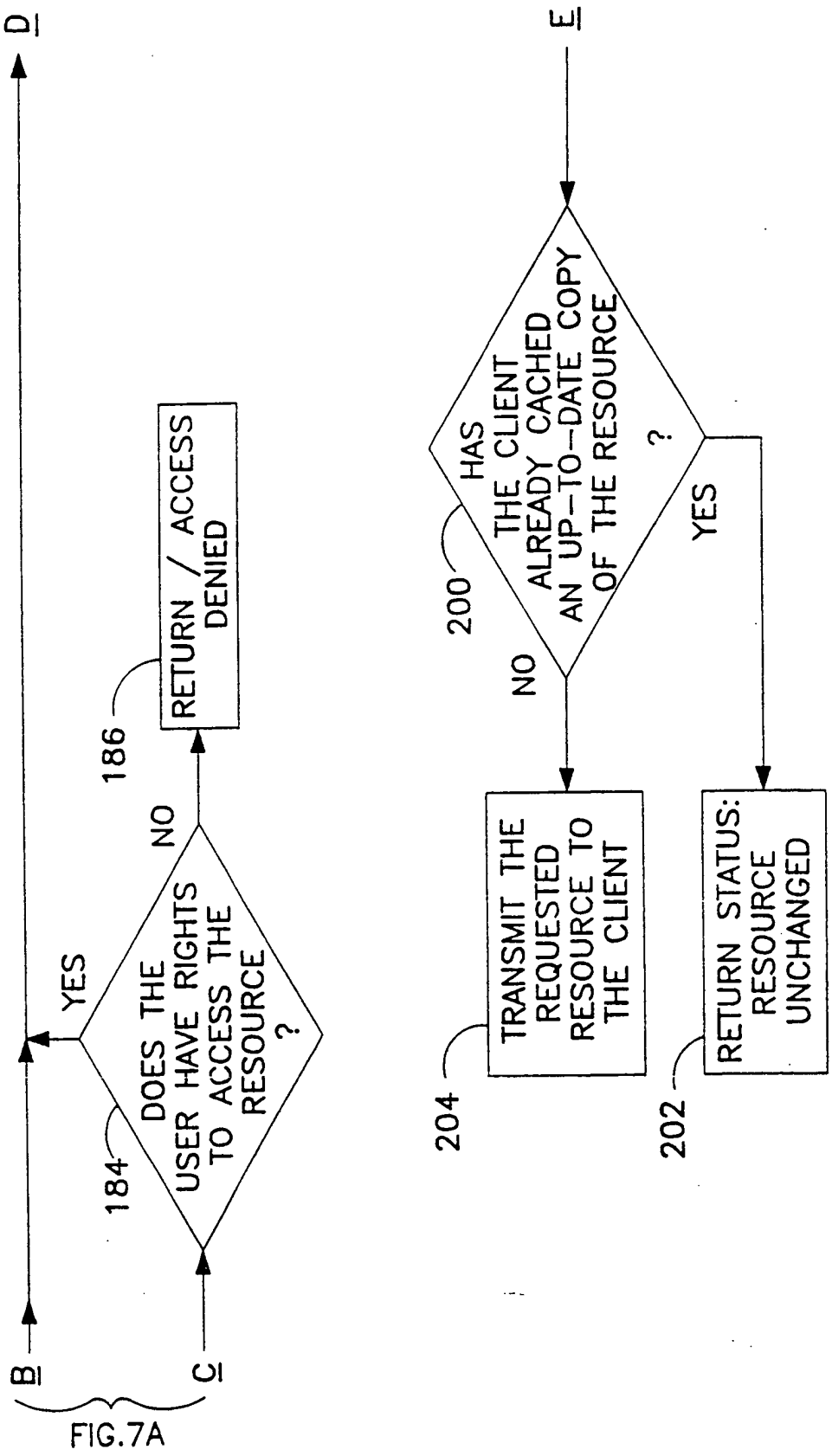


FIG.7A

FIG.7B

BNSDOCID. <WO _9907007A2_I_>

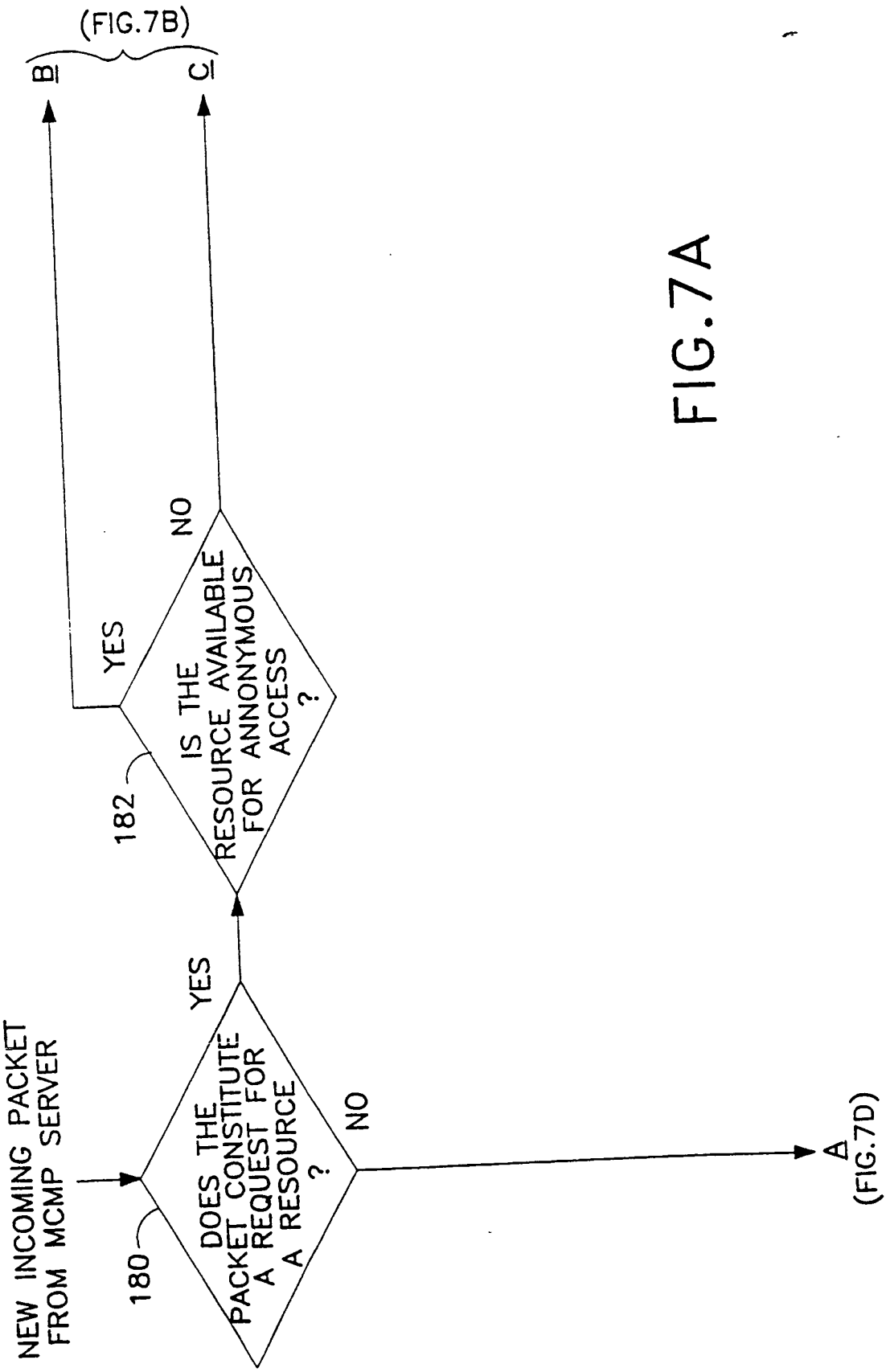


FIG.7A

BNSDOCID <WO 9907007A2 I >

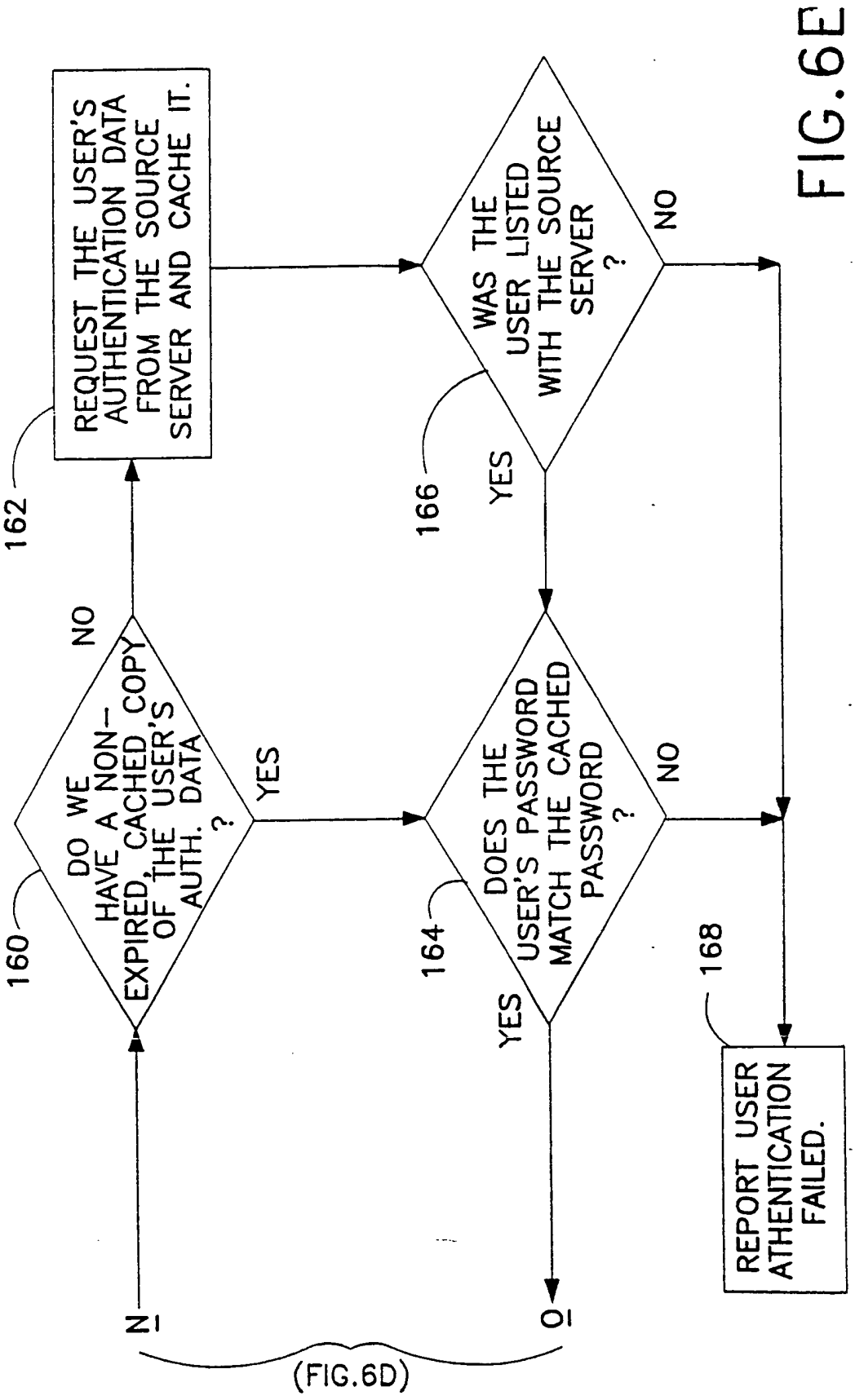


FIG.6E

BNSDOCID <WO_ 9907007A2 I >

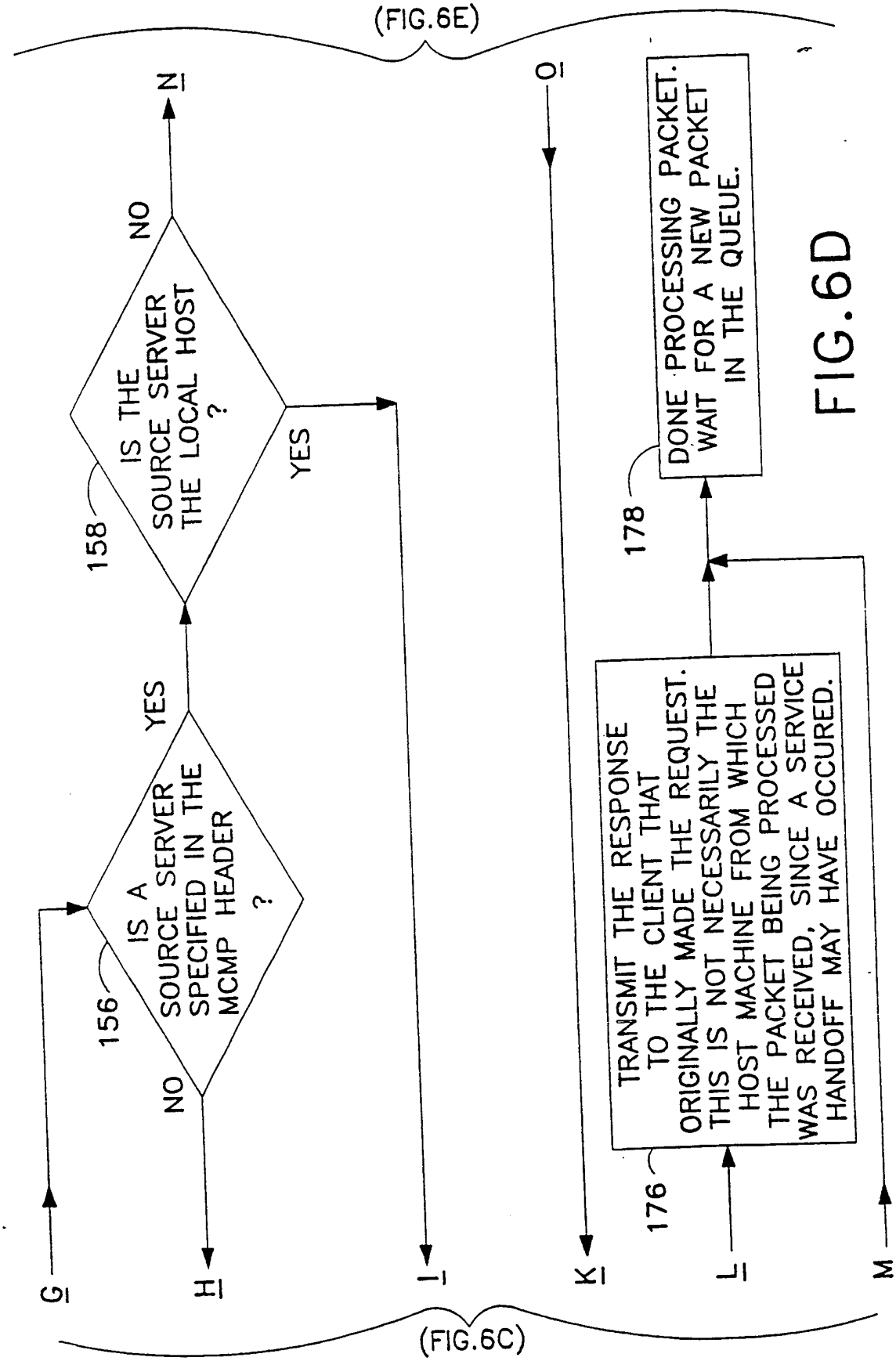


FIG.6D

ENSDOCID <WO 9907007A2 1_>

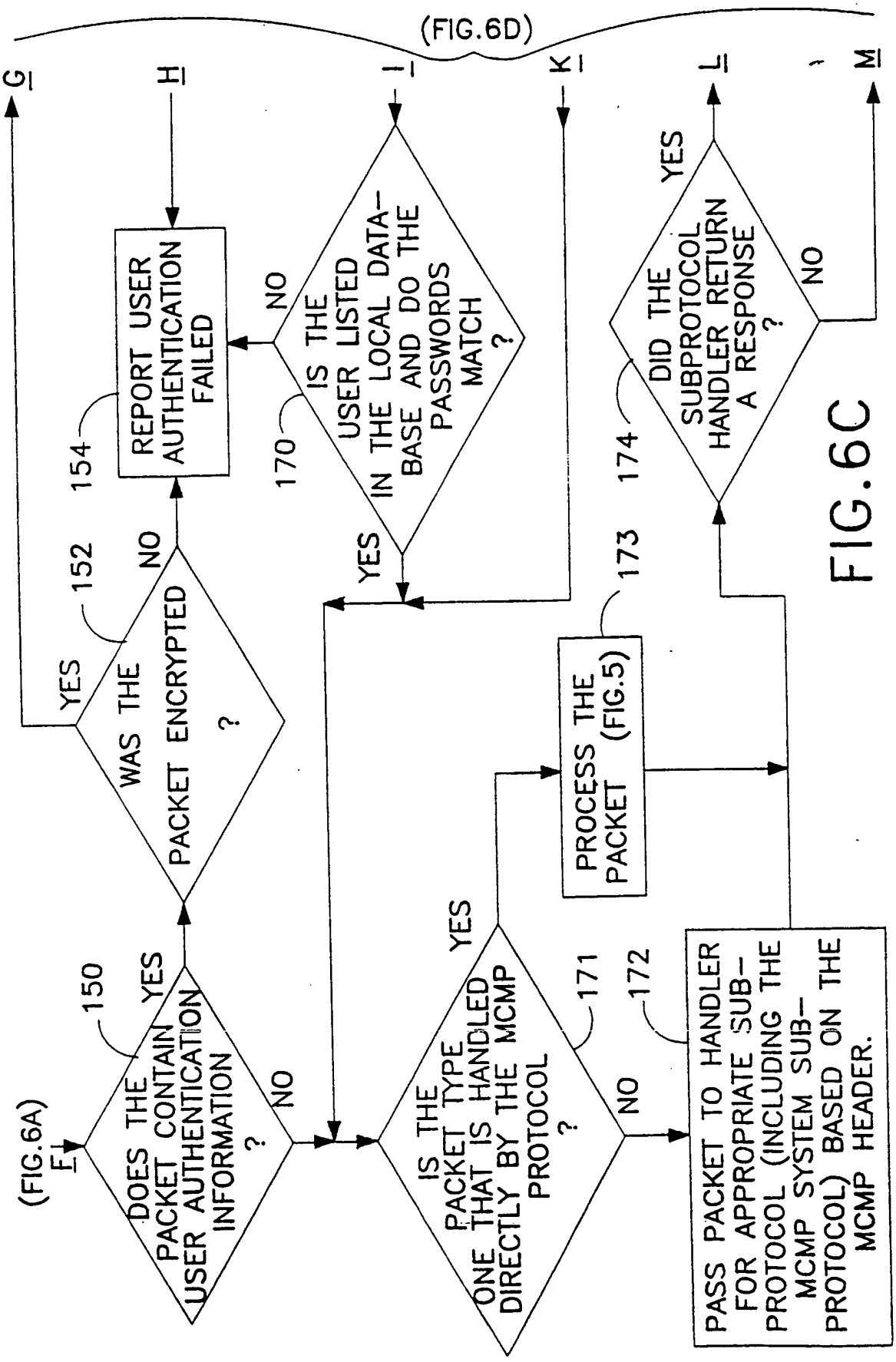
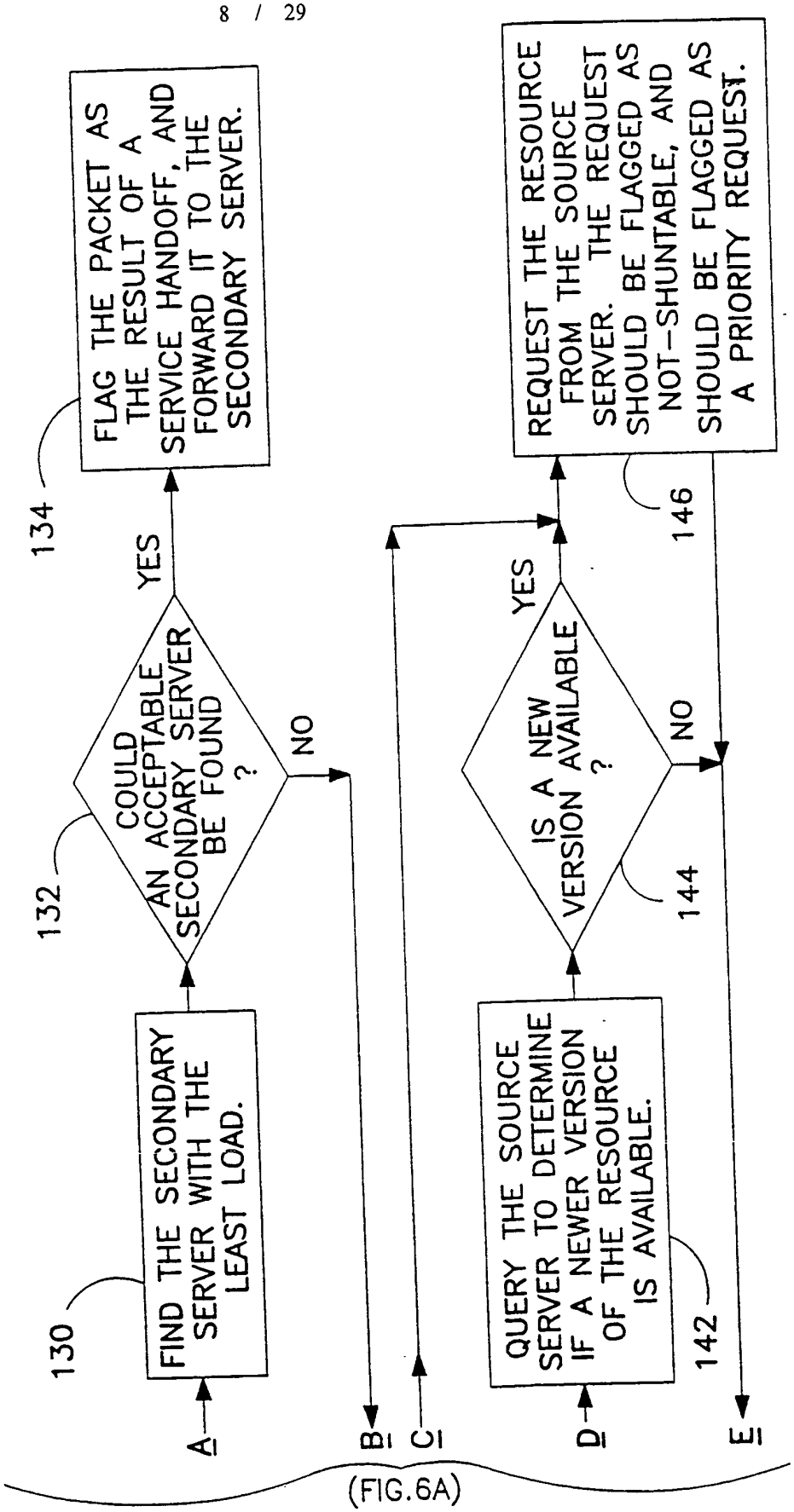


FIG.6C

SNSDOCID <WO 9907007A2 I >

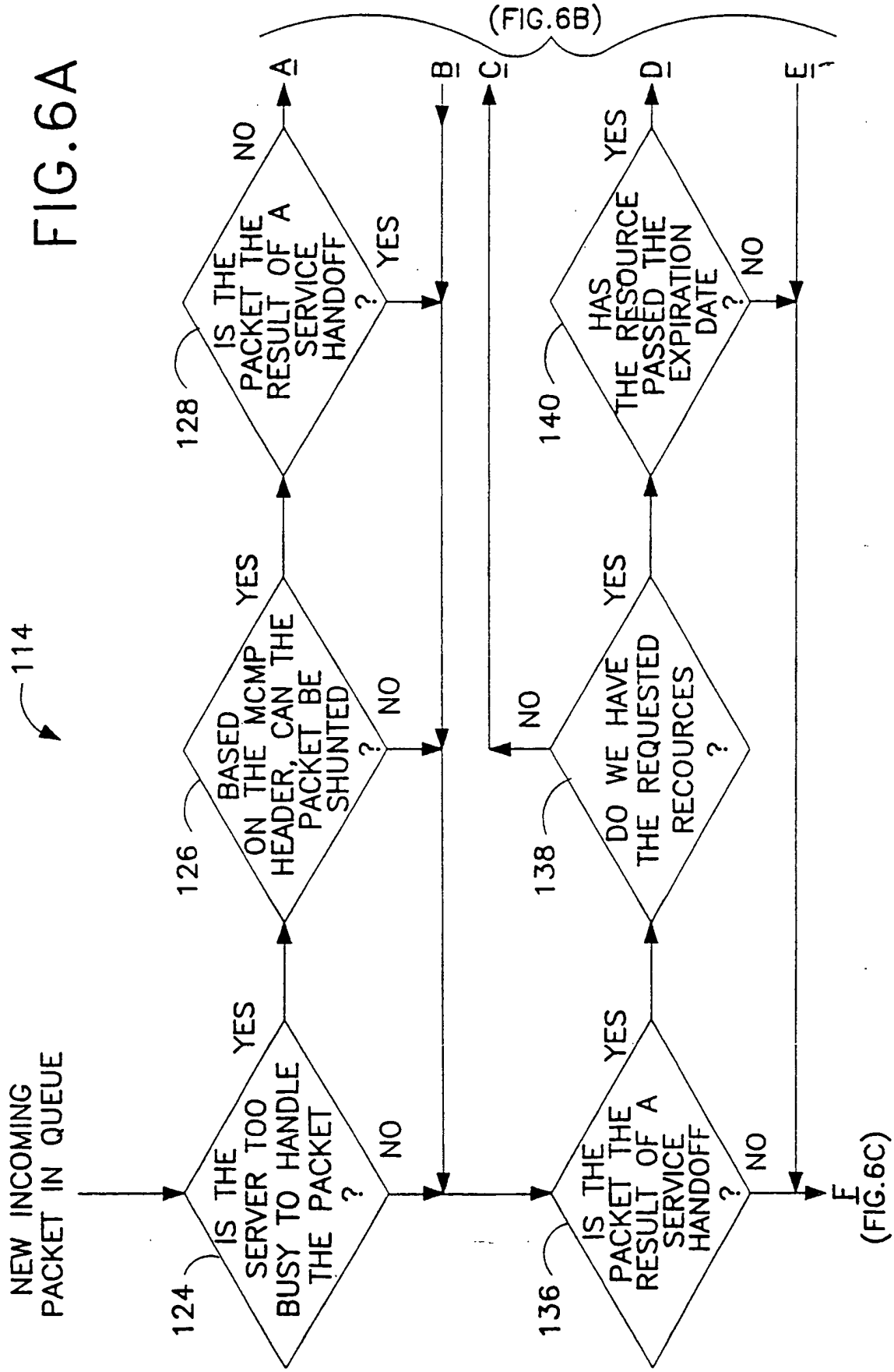
FIG.6B



(FIG.6A)

BNSDOCID <WO_ 9907007A2 I >

FIG.6A



BNSDOCID: <WO__9907007A2 I >

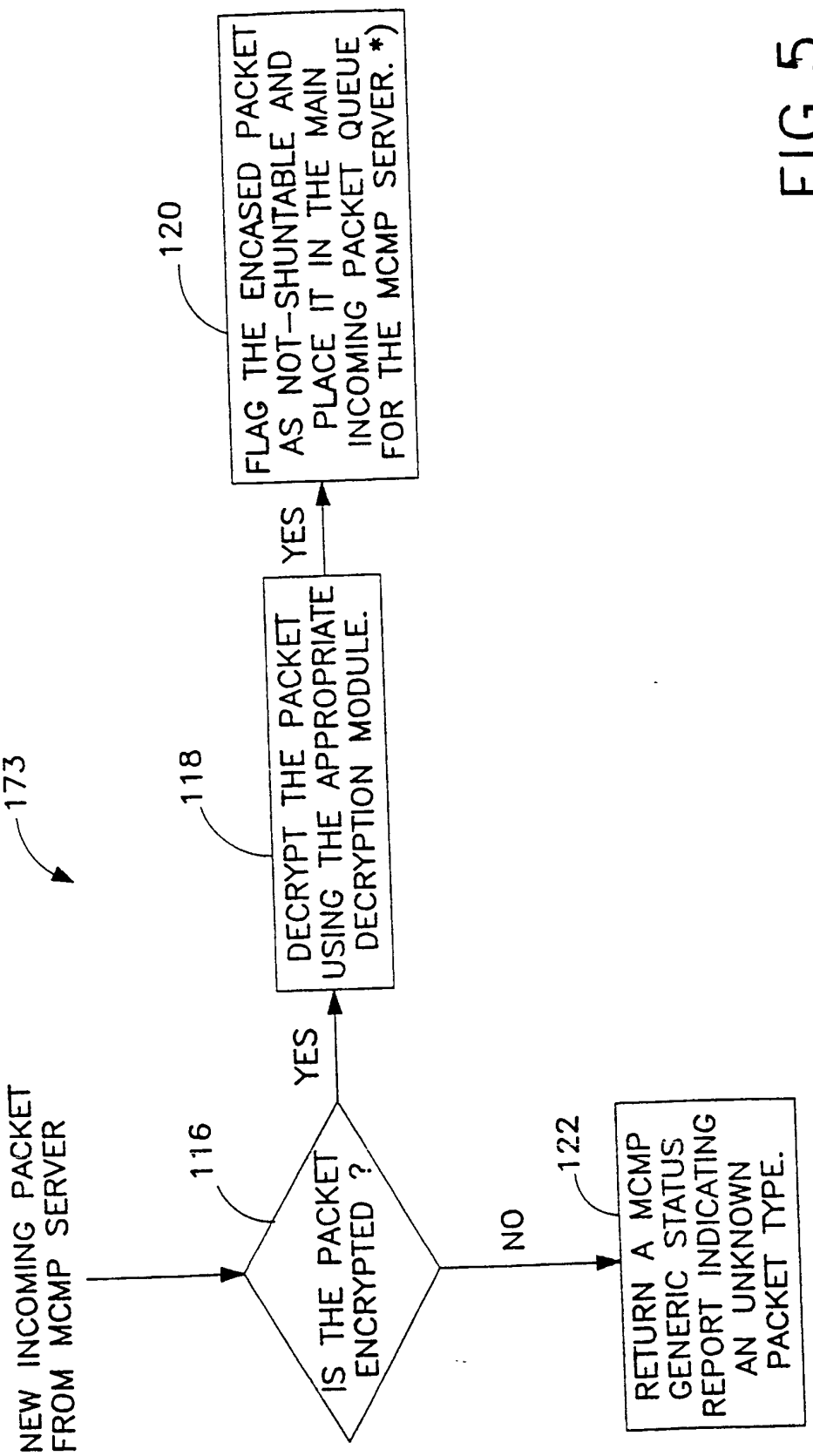


FIG.5

BNSDOCID <WO 9907097A2 I >

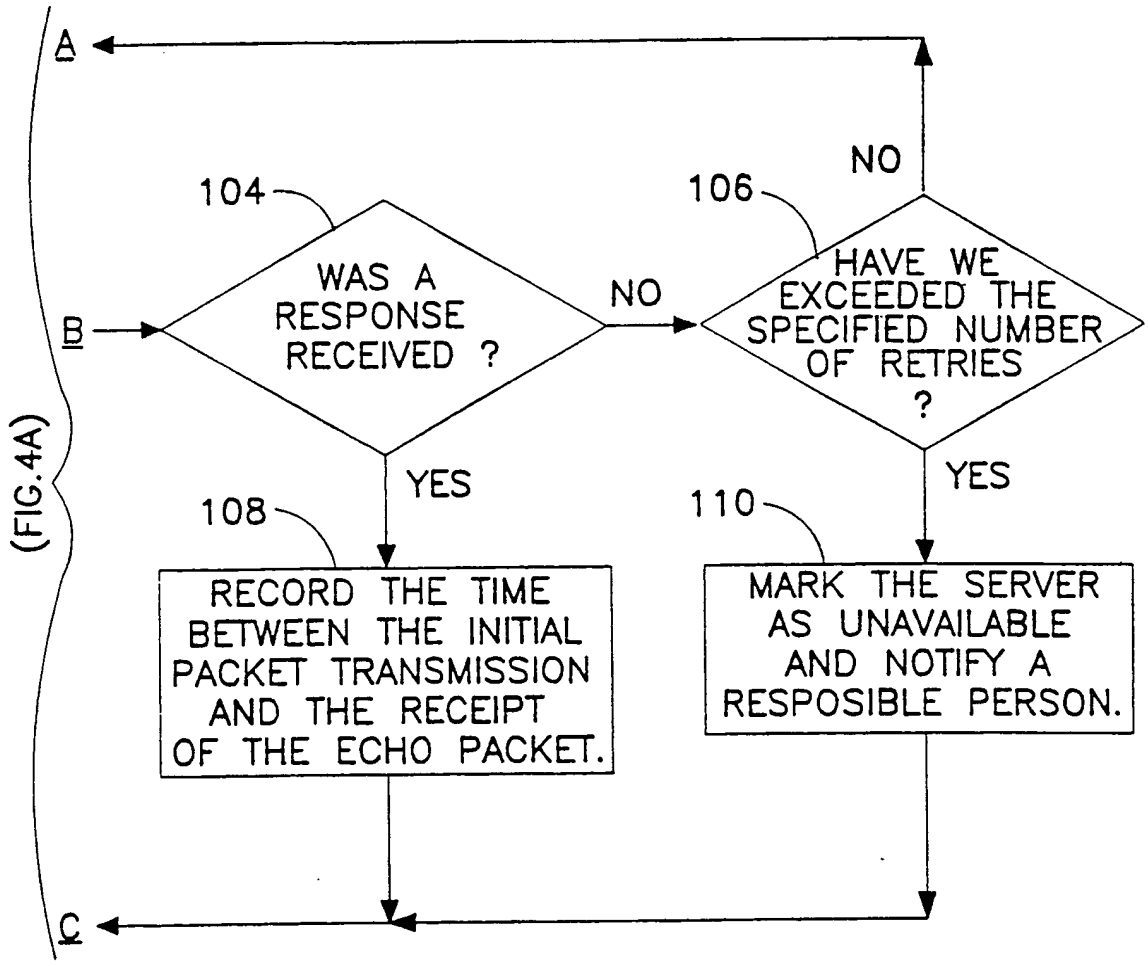


FIG.4B

BNSDOCID: <WO___9907007A2_I_>

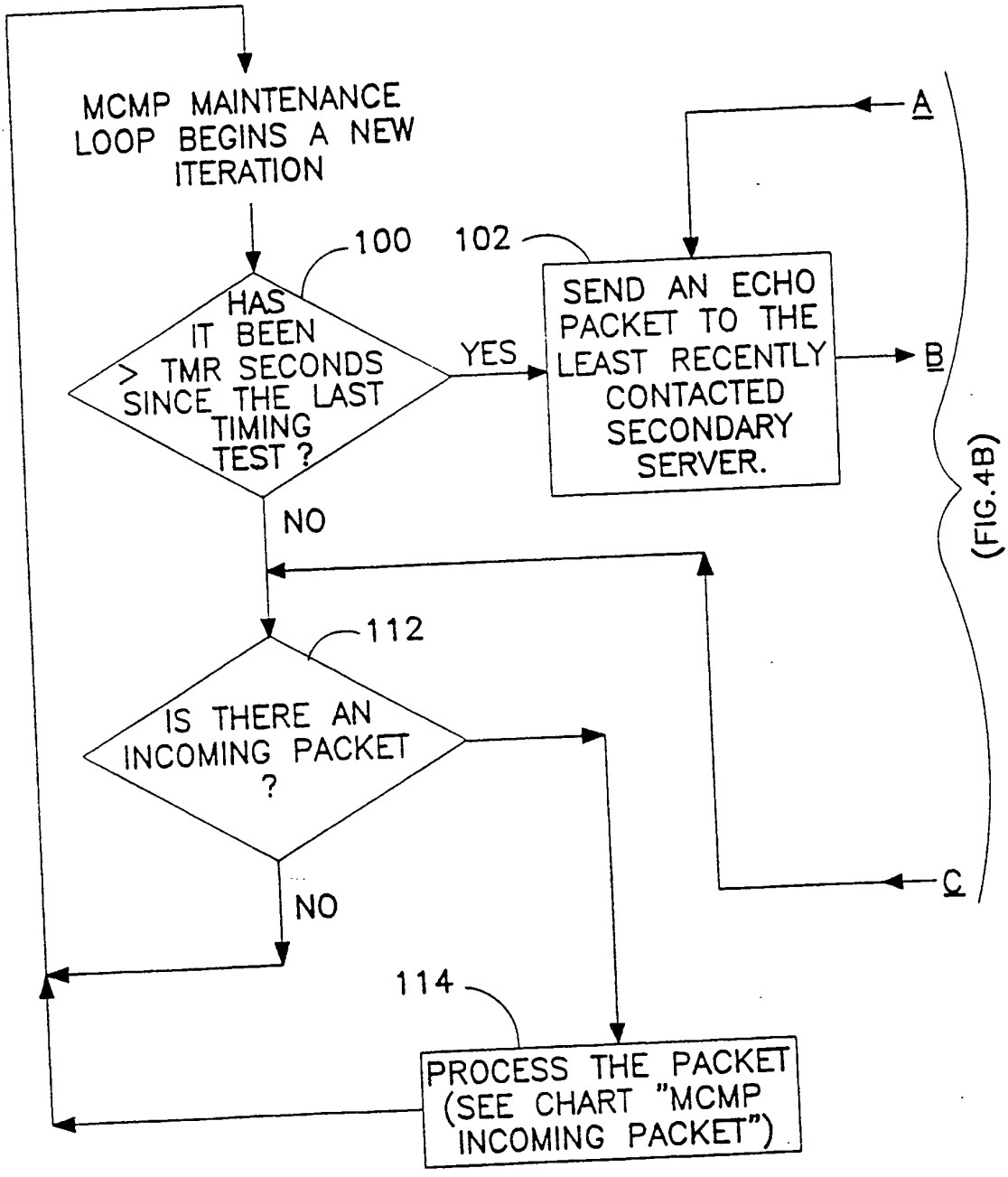
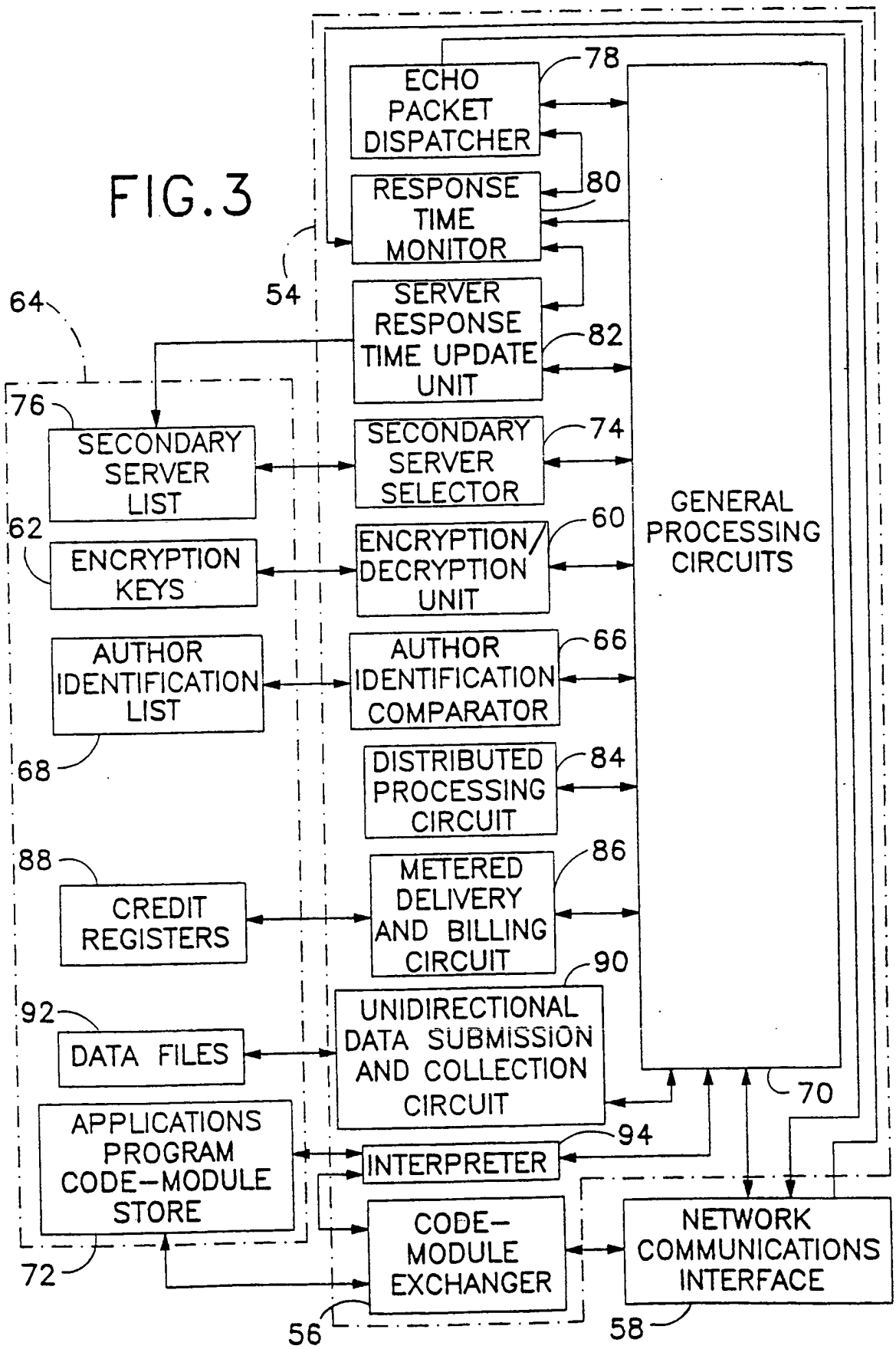


FIG.4A

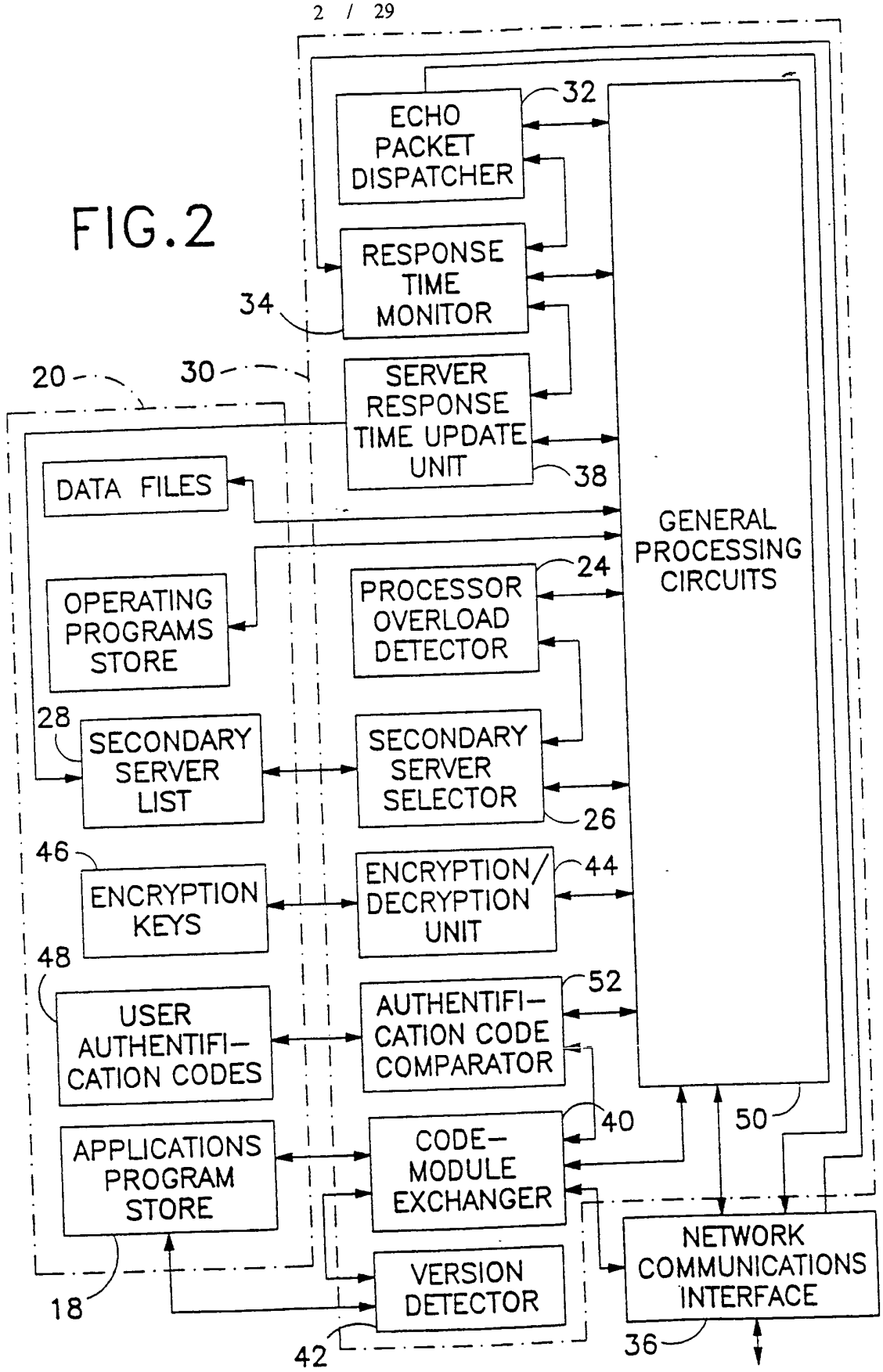
BNSDOCID <WO _ 9907007A2 I >

FIG.3



BNSDOCID: <WO_ _9907007A2_I_>

FIG.2



ENSDOCID <WO 9907007A2_I >

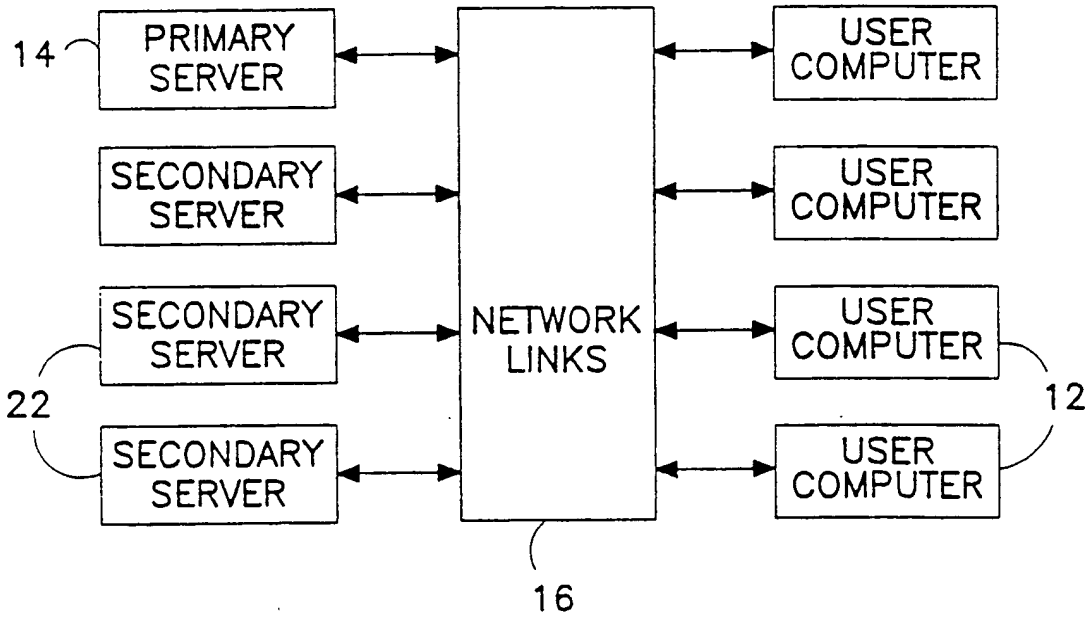


FIG. 1

SNSDOCID: <WO___9907007A2 I >

computer over said network link, the transmitted code modules being less than all of the code modules of said applications program; and

operating said second computer to follow programming instructions in the transmitted code modules to assist said first computer in executing its work load.

packets and receiving of responses to said echo packets from the respective servers, and (III) updating the response times in said list in accordance with the measured delays.

34. The computing system defined in claim 30, further comprising a software modified circuit operatively connected to said code module exchange means for encrypting communications transmitted to said remote computer and for decrypting communications received from said remote computer.

35. The computing system defined in claim 30 wherein said machine-executable code modules are written in a user-friendly programming code, further comprising an interpreter for translating said programming code into machine code directly utilizable by said processing circuitry.

36. A method for distributing processing among computers on a computer network, comprising:

storing an applications program in a nonvolatile memory of a first computer, said applications program being stored as a plurality of individual and independent machine-executable code modules;

executing portions of said applications program on said first computer;

transmitting a request over a network link from said first computer to a second computer not running at full capacity, said request being to take over the work load of said first computer;

in response to a request from said second computer, selectively transmitting machine-executable code modules of said applications program from said first computer to said second

digital processing circuitry operatively connected to said memory for executing at least one of said machine-executable code modules;

a communications link for communicating data and programs over a network to a remote computer; and

a code module exchange means operatively connected to said memory and to said communications link for communicating with a remote computer via a network link to obtain from said remote computer a further machine-executable code module of said applications program, said digital processing circuitry being operatively tied to said code module exchange means for executing said further machine-executable code module upon reception thereof from said remote computer.

31. The computing system defined in claim 30 wherein said computing system is a user computer on said network and said remote computer is a server computer.

32. The computing system defined in claim 31 wherein said memory contains a list of servers on said network, said list including response times for the respective servers, further comprising server selection means operatively connected to said memory and said code module exchange means for instructing said code module exchange means to communicate with a server selected from among said secondary servers as having a shortest response time, said remote computer being the selected server.

33. The computing system defined in claim 32, further comprising updating means operatively connected to said memory and said communications link for (I) periodically sending echo packets to said servers, (II) measuring delays between the sending of said echo

code module from among said machine-executable code modules from said first computer to said second computer.

27. The computing system defined in claim 26 wherein said first computer is a primary server and said second computer is a secondary server on said network, said first computer including detection means for detecting an overload condition of said first computer, said first computer further including shunting means operatively connected to said detection means and the communications link at said first computer for shunting an incoming user request to said second computer when said overload condition exists at a time of arrival of said user request.

28. The computing system defined in claim 27 wherein said first computer further includes updating means operatively connected to said memory and said communications link for (I) periodically sending echo packets to a plurality of secondary servers on said network, (II) measuring delays between the sending of said echo packets and a receiving of responses to said echo packets from the respective secondary servers, and (III) updating the response times in a list in said memory in accordance with the measured delays.

29. The computing system defined in claim 26 wherein said first computer is a server computer and said second computer is a user computer.

30. A computing system comprising:

a memory storing a portion of an applications program, said applications program comprising a plurality of individual and independent machine-executable code modules, only some of said machine-executable code modules being stored in said memory;

NSDOCID: <WO 9907007A2:1 >

communications link and said comparison means for decrypting said encryption packet prior to the comparing of said user authentification code in said request with said list of user authentification codes in said memory.

24. The computing system defined in claim 16, further comprising means for determining whether a requested code module has an updated version and for responding to said request with an invitation to said remote computer to accept the updated version of the requested code module.

25. The computing system defined in claim 1 wherein said machine-executable code modules are written in a user-friendly programming code, further comprising an interpreter for translating said programming code into machine code directly utilizable by said processing circuitry.

26. A computing system comprising:

a first computer;

a second computer remotely located relative to said first computer;

communications links at said first computer and said second computer for tying said first computer and said second computer to one another over a network,

said first computer including a nonvolatile memory storing at least a portion of an applications program, said applications program including a plurality of individual and independent machine-executable code modules,

each of the computers being provided with code module exchange means for cooperating with the code module exchange means of the other computer to transfer a single

request.

19. The computing system defined in claim 18 wherein the secondary server to which said user request is shunted is said remote computer, said single code module being required for enabling said remote computer to process the user request.

20. The computing system defined in claim 18, further comprising updating means operatively connected to said memory and said communications link for (I) periodically sending echo packets to said secondary servers, (II) measuring delays between the sending of said echo packets and a receiving of responses to said echo packets from the respective secondary servers, and (III) updating the response times in said list in accordance with the measured delays.

21. The computing system defined in claim 17 wherein said network is the Internet.

22. The computing system defined in claim 16 wherein said memory contains a stored list of user authentification codes, further comprising comparison means for comparing a user authentification code in said request with said list of user authentification codes in said memory and for preventing code-module retrieval and transmission in the event that the user authentification code in said request fails to correspond to any user authentification code in said list.

23. The computing system defined in claim 22 wherein said request from said remote computer is contained in an encryption packet, further comprising means connected to said

16. A computing system comprising:

digital processing circuitry;

a nonvolatile memory storing general operations programming and an applications program, said applications program including a plurality of individual and independent machine-executable code modules, said memory being connected to said processing circuitry to enable access to said memory by said processing circuitry;

a communications link for communicating data and programs over a network to a remote computer; and

a code module exchange means operatively connected to said memory and to said communications link for retrieving a single code module from among said machine-executable code modules and transferring said single code module to said remote computer in response to a request for said single code module from said remote computer.

17. The computing system defined in claim 16 wherein said computing system is a server computer on said network.

18. The computing system define in claim 17 wherein said memory contains a list of secondary servers on said network, said list including response times for the respective secondary servers, further comprising:

detection means for detecting an overload condition of the computing system; and

server selection means operatively connected to said detection means, said memory and said communications link for determining which of said secondary servers has a shortest response time and for shunting an incoming user request to the secondary server with the shortest response time when said overload condition exists at a time of arrival of said user

a machine-executable code module by said first computer, whereby said first computer and said second computer engage in interactive processing via said network.

12. The method defined in claim 8 wherein said machine-executable code modules are written in a user-friendly programming code, further comprising translating said selected one of said code module at said second computer from said programming code into machine code utilizable by said second computer.

13. The method defined in claim 8 wherein said machine-executable code modules each incorporate an author identification, further comprising, in response to an instruction received by said first computer over said network and prior to executing said one of said machine-executable code modules on said first computer, determining whether the particular author identification incorporated in said one of said machine-executable code modules is an allowed identification and proceeding with the executing of said one of said machine-executable code modules only if said particular author identification is an allowable identification.

14. The method defined in claim 8 wherein the storing of said portion of said applications program in said first computer includes caching said code modules in a nonvolatile memory of said first computer.

15. The method defined in claim 8, further comprising transmitting a request from said first computer to said second computer for a machine-executable code module during an idle time on said first computer.

3NSDOCID: <WO___9907007A2_I_>

9. The method defined in claim 8, further comprising:

sending a request from the first computer to a further computer for a list of servers on said network;

after transmission of said list of servers from said further computer to said first computer, determining response times for said servers by (a) sending echo packets from said first computer to said servers and (b) measuring, at said first computer, delays between the sending of said echo packets and a receiving of responses to said echo packets from the respective servers; and

selecting said second computer from among said servers as the server having the shortest response time, the transmitting of said request for said further machine-executable code module being executed after the selecting of said second computer.

10. The method defined in claim 8 wherein said request from said first computer is a second request directed to said second computer from said first computer, further comprising transmitting a first request from said first computer to said second computer via said network, said first request being for a first version of a particular machine-executable code module of said applications program, said second request being transmitted in response to a signal from said second computer indicating that a more recent version of said particular machine-executable code module is available, said second request being for said more recent version of said particular machine-executable code module.

11. The method defined in claim 8 wherein said second computer stores at least some of said machine-executable code modules of said applications program, said second computer executing at least one of said machine-executable code modules in response to the execution of

code module included in said machine-executable code modules;

transmitting, from said first computer to said second computer via said network link, a signal indicating that a more recent version of said particular code module is available, said selected one of said machine-executable code modules being said more recent version of said particular code module.

7. The method defined in claim 1 wherein said machine-executable code modules are written in a user-friendly programming code, further comprising translating said selected one of said code module at said second computer from said programming code into machine code directly utilizable by said second computer.

8. A method for optimally controlling storage and transfer of computer programs between computers on a network to facilitate interactive program usage, comprising:

storing a portion of an applications program in a first computer, said applications program comprising a plurality of individual and independent machine-executable code modules, only some of said machine-executable code modules being stored in said first computer;

executing at least one of said machine-executable code modules on said first computer;

transmitting, to a second computer via a network link, a request for a further machine-executable code module of said applications program;

receiving said further machine-executable code module at said first computer from said second computer over said network link; and

executing said further machine-executable code module on said first computer.

packets from said first computer to said secondary servers and (b) measuring, at said first computer, delays between the sending of said echo packets and a receiving of responses to said echo packets from the respective secondary servers; and

selecting said second computer from among said secondary servers as the secondary server having the shortest response time.

4. The method defined in claim 1, further comprising:

storing a list of user authentification codes in said memory;

upon receiving said request from said second computer, comparing a user authentification code in said request with said list of user authentification codes in said memory;

proceeding with the retrieving and transmitting of said selected one of said machine-executable code modules only if the user authentification code in said request matches a user authentification code in said list.

5. The method defined in claim 4 wherein said request from said second computer is contained in an encryption packet, further comprising decrypting said encryption packet prior to the comparing of said user authentification code in said request with said list of user authentification codes in said memory.

6. The method defined in claim 1 wherein said request from said second computer is a second request directed to said first computer from said second computer, further comprising:

receiving a first request directed to said first computer from said second computer via said network link, said first request asking for transmission of a first version of a particular

WHAT IS CLAIMED IS:

1. A method for optimally controlling storage and transfer of computer programs between computers on a network to facilitate interactive program usage, comprising:

storing an applications program in a nonvolatile memory of a first computer, said applications program being stored as a plurality of individual and independent machine-executable code modules;

in response to a request from a second computer transmitted over a network link, retrieving a selected one of said machine-executable code modules and only said selected one of said machine-executable code modules from said memory; and

transmitting said selected one of said machine-executable code modules over said network link to said second computer.

2. The method defined in claim 1 wherein said first computer is a server computer on a network, said second computer being a secondary server on said network, further comprising, in response to a user request directed to said first computer, forwarding said user request from said first computer to said second computer to initiate processing of said user request by said second computer, said selected one of said machine-executable code modules being required by said second computer to process said user request.

3. The method defined in claim 2, further comprising:

storing, in a memory of said first computer, a list of secondary servers on said network, said list including response times for the respective secondary servers;

periodically updating said response times, said updating including (a) sending echo

Although the invention has been described in terms of particular embodiments and applications, one of ordinary skill in the art, in light of this teaching, can generate additional embodiments and modifications without departing from the spirit of or exceeding the scope of the claimed invention. Accordingly, it is to be understood that the drawings and descriptions herein are proffered by way of example to facilitate comprehension of the invention and should not be construed to limit the scope thereof.

zero

Zeroes a variable or a specified number of bytes.

zero variable [,length]

Description

Zeroes bytes of variable space. The one-tag form zeroes only the variable specified. The two-tag form zeroes any number of bytes, beginning at the specified *variable* ; variable boundaries can be crossed with this form.

Example

To zero a large variable with one command.

```
define   global
var, 128    $$ a 128-byte buffer
define   end
*
      zero   var    $$ zero all 128 bytes]
```

colon :

comma ,

end-of-line ↵

universal delimiter |

Examples

Example 1

The variable *month* is used to selectively pick the text for displaying a month. The first two cases (negative and zero) are not used for text.

```
write    Today's month is $$$ retain the trailing space

writec   month,,,January,February,March,April,May,June,
         July,August,September,October,November,December
```

Example 2

Multiple lines of text can occur in each selective case. Delimiters not used as the selector delimiter can occur in the text.

```
writec   question;;;
         What city was once called Fort Dearborn?

         Hint: it is also called the Windy City.;      $$ case 1
         What is the largest city in the US?

         Hint: it is on the east coast.;         $$ case 2
         What city is known as the Gateway to the West?

         Hint: it is on the Mississippi.;        $$ case 3
         ...
```

Example 2

Embedded show commands can display both text and numbers from variables. The following would display something like:

Bill:

your score is *92%*

the class average was *85%*.

```
write    «showa, name»:
         your score is «show, score»%
         the class average was «show,average»%.
```

Example 3

```
write    Special effects such as underlining and italics are easily
         put on characters through insertion of embedded code sequences
         in the editors.  Some attributes such as size can use the
         embedded command form so that they «size,4» can be seen clearly
         in «color, red+» printouts of your code.writec
```

writec

Selectively display text.

---

writec SELECTOR; TEXT; TEXT;...

Description

The selective form of the **write** command.

The delimiter first following the *SELECTOR* is used to separate all following *TEXT* cases and can be a:

semicolon ;

text start" uncover code sequence Esc-&. A **writex** variant of this command does not save the starting text attributes: it restores attributes to those saved by the previous **write** command.

The display of the contents of variables and buffers can also be embedded in the text of a **write** by using the embedded form of **show** commands:

show | s	displays a variable or expression
showa | a	displays text in a buffer
showh | h	displays a buffer in hexadecimal format
showt | t	displays a variable in tabular format
showv | v	displays a text buffer including null codes

Trailing spaces on a line of text are normally stripped off by the compiler: they can be included by ending the line with a triple dollar sign comment ($$$).

Examples

Example 1

The following write goes on for three lines in source code. However, it will be displayed for many more lines on the screen since the margins as set by the **at** command make for a very narrow paragraph.

```
at      1:10; 10:30              $$ left margin is char 10
                                 $$ right margin is char 30
text    margin;wordwrap          $$ wrap full words at right margin
write   This is a paragraph of text that goes on and on.  It will
        appear nicely between the margins as set by the preceding at
        command. Continued lines of text in a write statement begin by
        tabbing over the command field.
```

write

Displays text.

write *TEXT*

Description

Displays text on the screen beginning at the current screen location. The location is usually set by a preceding **at** command which can also set left and right text margins. The text of the **write** can extend for many lines whose wraparound behavior is controlled by the **text margin** command.

Text attributes (e.g., color, underline, drop shadow, font design, etc.) can be changed character-by-character through use of embedded text attributes. These can occur in the text as: uncover code sequences that are inserted through menu options in the source code and display editors; and embedded commands that are included through use of the embed symbols « ». Over 50 text attribute uncover codes exits . The commands and **text** command keywords that can embed attributes are:

color | c	selects foreground display color
colore | ce	selects background display color
margin | ma	controls text wrapping
mode | m	selects how text plots over existing displays
size | s	changes size of fonts
spacing | sp	selects proportional or fixed spacing

The text attributes that exist at the start of a **write** are saved and restored at the end of the **write** so that any attribute changes made within the text do not extend to following commands. These starting saved attributes can be restored to at any time within the text by use of the "status

SNSDOCID: <WO. 9907007A2_I_>

is saved in the memory pool. When window compression is set **on**, the saved data occupies considerably less memory pool space; when compression is set **off**, more space is used in the memory pool but on older model computers the window may open and close more quickly.

System Variables

zreturn

-1	Operation successful
18	Insufficient memory in memory pool
23	Part of window off physical screen

Miscellaneous

The following system variables are affected by window

zdispx	width and height of window
zdispy	
zxmax	maximum x- and y-coordinates in window
zymax	
zwindowx	absolute x- and y-coordinates of lower left window corner
zwindowy	
zclipx1	absolute x- and y-coordinates of lower left clipping area
zclipy1	
zclipx2	absolute x- and y-coordinates of upper right clipping area
zclipy2	

window frame [; LOCATION [; LOCATION]]

Changes the boundaries of the current display window. The contents of the window and the saved display under the window are not changed. **window frame** can be used on the outer-level physical screen window.

window frameabs [; LOCATION [; LOCATION]]

Identical to **window frame** except that the rectangular location is interpreted as though **enable absolute** is in effect: coordinates refer to the physical screen. A previously opened window or the **origin** command have no effect.

window relative; [on | off | CONDITION]

Turns **on** or **off** the use of window-relative coordinates for the current display window. The default when a window is opened is relative on. If relative coordinates are turned **off**, then coordinates are relative to the physical screen rather than the current window; however, **scale, rotate, origin** and **window** clipping still apply.

For the *CONDITIONAL* form, an expression that evaluates to a negative value is treated as **on** while zero or a positive value is treated as **off.**

window clip; [on | off | CONDITION]

Turns **on** or **off** the clipping of display objects that extend outside the current window. The default when a window is opened is **clip on**. If clipping is turned **off**, objects extending outside the current window will be displayed.

The system variables **zclipx1, zclipy1, zclipx2, zclipy2** hold the absolute coordinates for the lower left and upper right of the clipping area.

window compress; [on | off | CONDITION]

Controls data compression of the underlying screen. The default when a window is opened is **compress on.** When a window is opened, the contents of the display under the window

```
Press any key to continue...
pause    pass=all
window   close
```

Example 2

```
at       5:15
window   open; 20,5
```

A window 20 characters wide and 5 lines deep is opened at the underlying screen (or window) location of line 5 character 15.

window openabs [; LOCATION [; LOCATION]]

Identical to **window open** except that the rectangular location is interpreted as though **enable absolute** is in effect: the *LOCATION* coordinates refer to the physical screen and are not affected by any previously open window or the **origin** command.

window close [; noplot] [; noarea]

Closes the current display window, restoring from the memory pool the saved display beneath the window, the display **status**, the **area** definitions, and the status of **window relative**, **window clip**, and **window compress**. The optional keyword **noplot** inhibits the restoration of the display beneath the window. The optional keyword **noarea** inhibits the restoration of area defines.

window reset [; noplot] [; noarea]

Closes all open windows in order from the last to the first opened, restoring the display beneath the window, the display **status**, the **area** definitions, and the status of **window relative**, **window clip**, and **window compress.** The optional keyword **noplot** inhibits the restoration of the display beneath the window. The optional keyword **noarea** inhibits the restoration of **area** defines.

the first character on the top line in the window. Display objects are clipped if they extend beyond the window (see **window clip**).

The rectangular coordinates for **window** are affected by any previously opened window and the **origin** command; they are not affected by **scale, rotate**, or **window clip.** Use **window openabs** or **enable absolute** to locate a window relative to the physical screen without regard to any previously opened window. A window is not opened if it would extend off the physical screen (**zreturn** = 23).

When both rectangle *LOCATIONS* are blank, the entire screen or previous **window** is implied. When only one rectangle *LOCATION* (the first) is present, it denotes that the current screen location should be used for the starting position and *LOCATION* gives the number of characters or characters and lines of the currently selected font to size the window by.

Example 1

A window opens on a VGA screen with a blue background, white border, and some text. When any key is pressed, the window is closed and the underlying display is restored.

```
window   open; 100,100; 400,250 $$ open window, middle of screen
colore   blue
erase                    $$ clear window to blue

color    white+

box      ;;1                  $$ put border around window edge
at       2:4
write    The author of this lesson is:

            Dr. John Burdeen
            University of Illinois at Urbana
```

close	closes current window, restores previous display status
reset	closes all windows, restores previous display status
frame	changes window boundaries using relative coordinates
frameabs	like **frame** except uses absolute coordinates
relative	controls use of window-relative coordinates
clip	controls clipping of graphics extending beyond window
compress	controls compression of saved underlying display

Description

The **window** command opens, closes, and alters display windows. When a window is opened, the current display status and screen display under the window is saved in the memory pool. Further display operations are relative to and appear only in the window unless the absolute coordinates feature is selected. When a window is closed, the previous display status and screen display under the window is restored.

On entry to the system, an initial window is opened that corresponds to the entire physical screen. Properties of this outer-level window can be altered with the **window** command but it cannot be closed.

window open [; LOCATION [; LOCATION]]

Opens a display window over the rectangular area specified, saving in the memory pool the screen display underneath the rectangle, the display status (see **status**), the current area settings (see **area**), and the current setting for relative coordinates (**window relative**), clipping (**window clip**) and compression (**window compress**). Upon closing the window, the saved data are used to restore the screen to its previous state.

Screen coordinates are relative to the newly opened display window: the graphic coordinate 0,0 is at the lower left corner of the window and the character coordinate 1:1 refers to

width

Description

Specifies a width for graphics; default is one pixel. System variable zwidth holds the current width value. Width is added symmetrically on both sides of the basic graphic elements except for dot which just grows larger.

The following commands are affected:

dot

draw

polygon (non-filled only)

circle (non-filled only)

ellipse (non-filled only)

box and erase are not directly affected by the width command. However, they can use the system variable zwidth for their width argument and thereby become connected. Note that these graphics only "widen" inwardly due to their pre-existing width argument definition.

The current width setting is part of the status buffer

```
width    expression
width    5        $$ following line graphics 5 pixels thick
width    var      $$ width depends on variable
```

window

Manages display windows.

---

window keyword...

open opens display window using relative coordinates

openabs like **open** except uses absolute coordinates

underline color	4	current underline color
(reserved)	6	
delay break	1	-1 = yes; 0= no
margin	1	-1 = advance; 0= wrap; 1 = release; 2= wordwrap
direction	2	0 = right; 90 = up; 180 = left; 270 = down
rotation	2	0..359
align	1	-1 = baseline; 0= bottom
delay	4.real	delay in seconds
reveal	1	0 = off; 1 = reveal* ; 2 = all; 3 =allcr; 4= editor
x increment	2	x - increment
y increment	2	y - increment

* for the obsolete **reveal on** command.

Example

Reads the current text attribute settings and displays them.

```
define   local
settings,54
.        sizex,2
.        sizey,2
.        bold,1
...
define   end
*
text     info; settings
write    The current X-size = «s, sizex»
              Y-size = «s, sizey»
         and bold is
writec   bold; on; off
```


BNSDOCID <WO 9907007A2 I >

off (0) return to functioning hidden codes

text measure

Initializes the text measuring system variables that can be used to determine the rectangular extent of a paragraph of text. The system variables define the corners of a rectangle:

(zplotxl, zplotyl) ; (zplotxh, zplotyh)

Example

A paragraph of any width and height is framed. Multiple **write** and **show** commands can occur where the example uses a single **write**.

```
text      measure
write     This is a paragraph
          of text...
box       zplotxl,zplotyl; zplotxh,zplotyh; 2
```

text info; infoBuffer [,length]

Reads information on the current settings of the text options into a buffer for the given length. If the length is not specified, the defined length of the buffer is used. The format of the buffer is:

Text Option	Bytes	Possible Values
size x	2	1...9
size y	2	1...9
bold	1	-1 = on; 0 = off
italic	1	-1 = on; 0 = off
narrow	1	-1 = on; 0 = off
(reserved)	5	
blink	1	-1 = on; 0= off (text.obsolete)
bright	1	-1 = on; 0= off (text.obsolete)
y base same	1	-1 = on; 0 = off (charsets, obsolete)
spacing	1	-1 = variable; 0= fixed
shadow	1	-1 = on; 0 = off
shadow color	4	current shadow color
(reserved)	5	
underline	1	-1 = on; 0= off
underline	1	-1 = foregnd; 0= specific color

BNSDOCID: <WO__9907007A2_I >

When **bottom** alignment is in effect, characters align along the bottoms of their character cells. When several sizes or styles of characters are lined up by their bottoms, they don't appear to be on a common line due to the need for the descender part to grow with the size of the character

When **baseline** alignment is in effect, the baseline in the **standard** font is used to determine the placement of characters from all other fonts on the screen. In the **standard** size 1 font, the baseline is located **zcharb** above the bottom of the character cell. For the currently selected font and size enlargement, the baseline is located **zfontb** above the bottom of the cell. Since the baselines are to line up, the current font is plotted with the bottom of its cell located **zcharb-zfontb** vertical dots from the set **zy** location where the **standard** font size 1 bottom plots.

The variable tag from uses the values: -1 = baseline, 0 = bottom

text **reveal; off | all | allcr | editor | reveal**

Controls display of hidden text codes; the default is **off**.

Displays rather than functions the hidden codes occurring in text such as uncover escape codes, superscripts, subscripts and the carriage return (CR). The hidden code display is either a single symbol or begins with a square followed by one to three additional characters specifying the hidden function.. The variable tag form use the values shown after the keywords.

all	(2)	displays hidden codes except CR; CR functions
allcr	(3)	displays hidden codes and CR; CR functions
editor	(4)	displays hidden codes and CR; CR does not function

starting point of the text around the screen with each iteration of the **loop**. The text in the **write** statement starts with a space which in mode **rewrite** removes the starting part of the underlying "W" from the previous iteration. If the example is changed to move to the left, the text should have the trailing space at the end.

```
zero     x
text     margin; wrap                    $$ turn on screen wrap-around
text     size; 2
mode     rewrite                  $$ replace screen underneath
flow     jump; %other; start; break      $$ provide exit on any key
loop
.        calcs    x = zxmax; x <= 0; x+1
.        at       x, zymax/2              $$ half-way up screen
.        write    Welcome to AVIATION BASICS $$note first space
endloop
```

text align; baseline | bottom | align

Aligns characters of different sizes or fonts; the default is **baseline**.

The **align** options control how characters of different sizes or fonts plot relative to each other. This is done by lining them up either on their baselines or the bottoms of the character cell. The baseline is usually the bottom point of letters without descenders and is set in the Font Editor for each font.

When **baseline** alignment is in effect, characters align along their baseline. Different sizes and styles of fonts appear to be on the same line:

editing **write** objects in the Display Editor. Several special characters such as hard-space and hard-hyphen exist for use in wordwrap mode.

The **wrap** option causes characters and parts thereof that extend past the edge of the window or screen to wrap around to the opposite side of the screen and continue plotting: any margins set by **at** are ignored. The wrap occurs even within a character so that pieces of it may appear on both sides of the screen or window. A billboard effect can be given to animated text that wraps around the screen.

The **release** option allows character plotting to continue or start outside the window or screen: any margins set by **at** are ignored. No advance is done. If the standard clipping is in effect, only the characters and parts thereof that exist in the window appear.

The variable tag form uses the values: -1 = advance, 0 = wrap, 1 = release, 2 = wordwrap.

Example 1

This paragraph of text
looks just like this when
plotted in **wordwrap** mode
by the following code.

```
screen   vga
text     margin; wordwrap
at       0,464;204, 0
write    This paragraph of text looks just like this when plotted in wordwrap
mode by the following code.
```

Example 2

The code below causes the text *Welcome to Aviation Basics* to cycle around the screen from left to right as if on a billboard. The **calcs** statement tests for the screen right edge and either advances the variable *x* by one or resets it to 0. The following **at** statement then advances the

The character grid is not rotated unless done by a previous **text rotate** statement.

The direction that characters will be displayed is specified by keywords (or variable value):

right	(0)	from left to right
left	(180)	from right to left
up	(90)	from bottom to top
down	(270)	from top to bottom

With direction set to **left,** various leftward plotting languages such as Arabic and Hebrew can be displayed when using the proper font.

Example

```
text     direction; left
text     delay; .1
write    drawtfel gniog
```

Causes the phrase *going leftward* to slowly appear on the screen starting from its right end.

text margin; advance | release | wrap | wordwrap | margin

Controls the behavior of text going past the right margin or the edge of the screen or window; the default is to **advance** character plotting to the next line.

The **advance** option is the default for displaying lines of text. It pre-checks whether any part of a character would plot past the right margin or edge of the window or screen and if so advances the character to the left margin on the next line.

The **wordwrap** option pre-checks whether any part of a word would plot past the right margin or edge of the window or screen and if so "wordwraps" the entire word to the left margin of the next line. This mode is commonly used for wordprocessing and can be accessed while

BNSDOCID: <WO___9907007A2_I_>

Example

Causes each successive character to be plotted 2 pixels to the right and 3 pixels lower than its normal position.

```
text      increment; 2,-3
write     Descending text!
```

text rotate; degrees

Rotates the character plotting baseline in degrees; the default is 0.

For best results, use rotations that are a multiple of 90 degrees on screens that have a 1:1 aspect ratio (generally type VGA or EVGA).

Example 1

Rotates the character plotting baseline by 90 degrees counter-clockwise so that the word *TEMPERATURE* can be the label for the vertical axis of a graph.

```
at        250,200
text      rotate; 90
write     TEMPERATURE
```

Example 2

Four copies of the alphabet appear rotated around the center of the screen.

```
loop      angle <= 0, 270, 90
.         at zxmax/2, zymax/2
.         text      rotate; angle
.         write     ABCDEFGHIJKLMNOPQRSTUVWXYZ
endloop
```

text direction; right | left | up | down | direction

Controls the direction that text will be displayed relative to the character plotting baseline; the default is **right**.

Example

Underlines the first sentence with the foreground color and the second sentence with blue. Using escape codes in the text is an alternate method for turning on underlining.

```
text      underline; on,foregnd
write     This text is underlined in the same color as the text.
text      underline; on, blue
write     This text is underlined in blue.
```

text delay; seconds [;break]

Delays for the specified time in seconds after plotting each character; the default is 0. Presence of the optional **break** modifier allows any break modified flow event to occur rapidly when a long delayed text statement is executing: the delay is temporarily turned off for the remainder of the output allowing the command to quickly complete so that the flow branch can occur.

Example

Causes the message to come on the screen slowly (a quarter second delay between each character).

```
text      delay; 0.25
text      size;2
color     red+
write     WARNING!
```

text increment; x,y

Adds an additional spacing offset after plotting a character; the default is 0,0.

After plotting a character, the *x* offset is added to the horizontal location and the *y* offset is added to the vertical location. Negative offset values are allowed.

1512

text shadow; on | off | CONDITION [,COLOR]

Controls character drop shadowing; the default is **off**. The shadow color defaults to **white.**

The shadow effect is produced by displaying each character twice: the first plotting is offset and in the shadow color while the second is in the foreground color. The drop shadow offset for a font is set in the Font Editor. If no shadow color is specified, the last specified color is used. The default shadow color is **white** after an **initial** command.

Example

Gives a very patriotic effect for an American display.

```
colore   blue
erase
color    white+
text     shadow; on,red+
text     size; 2
at       10:10
write    An American statement.
```

text underline; on | off | CONDITION [,COLOR | foregnd]

Controls character underlining, the default is **off**. The color of an underline defaults to the current foreground (keyword **foregnd**) color as used for plotting the character.

The location of the underline and the size of the descender gap is specified for a font in the Font Editor. If no underline color is specified, the last specified underline color is used. The default underline color is **foregnd** after an **initial** command.

With **variable** spacing, each character occupies only the horizontal space required. For example, the lower case *i* occupies less horizontal space than the upper case *M* Many more characters per line can be plotted with **variable** spacing than with **fixed.** Studies show that **variable** spacing increases reading speed.

The variable tag form uses the values: -1 = variable, 0= fixed.

Example 1

When used with the standard font, the second line of text with variable spacing takes up less space on the screen than the first line with fixed spacing.

```
at        10:1
text      spacing; fixed
write     This text runs to here.
at        11:1
text      spacing; variable.
write     This text runs to here plus.
```

Example 2

Variable spacing is used for the labels but fixed spacing solves the problem of lining up the numbers.

```
at        6:10
text      spacing; variable
write     Pork
          Beef
          Chicken
at        6:20
text      spacing; fixed
write     365
            4
```

```
*       in +initial block
initial
text    bold; on
text    spacing; variable
...
status  save; default
```

text italic; on | off | CONDITION

Controls the italic attribute; the default is **off**.

Example

The escape codes for turning italic on and off have been inserted around the word *amorous*. In the Source Editor, these are input by pressing [CTRL] [A] [I] and [CTRL] [A] [SHIFT] [I] The italics are either synthesized from the current font or, if a font group is active, come directly from the appropriate italic font in the group

```
write   The word amorous is derived from Latin.
```

text narrow; on | off | CONDITION

Controls narrow font usage from a font group; the default is **off**.

The **narrow** attribute is of use for low resolution screens like the CGA in order to produce the maximum amount of text on a screen. Narrow fonts are supplied in the system standard font group to provide for upward compatibility with previous versions of the TenCORE system. **narrow** cannot be synthesized.

text spacing; fixed | variable | spacing

Controls character spacing; the default is **fixed.**

With **fixed** spacing, each character occupies the same horizontal space defined as the maximum character width for a font. It is often used to align tables of numbers or to produce emphasis.

SUBSTITUTE SHEET (RULE 26) <WO 9907007A2 I >

Example

A 4-by-4 matrix of X's appears showing the 16 sizes of character enlargement that can be synthesized for the two argument form of **size**. *letters* should be a font and not a font group otherwise the four *sizeX* = *sizeY* entries would attempt to match and use a group font that would not match in style the other synthesized sizes.

```
font     'letters'
loop     sizeX <= 1,4
.        loop     sizeY    <=1,4
.        .        at       4*sizeX+1: 4*sizeY+1
.        .        text     size; sizeX,sizeY
.        .        write    X
.        endloop
endloop
```

text **bold; on | off | CONDITION**

Controls the bold attribute; the default is **off**.

Example 1

The font *big* is made bold by having each pixel in a character horizontally duplicated.

```
font     big'     $$ a font block
text     bold; on
write    This text is synthesized bold.
```

Example 2

In the **+initial** control block, the **initial** command sets all the plotting parameters including the font to the system standard. Several text attributes are then changed and saved in the **default** status buffer. Any **jump** branch now resets to these default text attribute settings. Since the system standard font is a font group, the **bold** attribute selects a bold font from this group.

system variables **zfontf** and **zfontret** for more information on how fonts are selected from a font group to best match the attribute settings.

text size; size

Controls the size attribute; the default is **size 1**.

For a font, up to a 4-time enlargement of the base size character is synthesized in both the horizontal and vertical dimensions. For a font group, the system attempts to select an appropriately sized font in the group up to a maximum size of 9. If the size doesn't exist as a font in the group, sizes up to 4 are synthesized while larger sizes default to 1.

Example 1

If used with a font, character doubling from the base font is synthesized. If used with a font group, the appropriate size 2 font in the group is used; if none exists, it is synthesized from the base size 1 font.

```
text     size; 2
```

Example 2

Size is one of the text attributes that can be embedded into a line of text by use of the « and » embed symbols. The embed method has advantages over escape codes in that the text is not altered for viewing in the Source Editor and makes for a better printout of the coding.

```
write    «size,1»Start out small, «size,2»end up big.
```

text size; sizeX,sizeY

Sizes the horizontal and vertical character enlargement independently up to the maximum size value of 4. For unequal X and Y values, the characters are always synthesized. For equal X and Y values, the characters either come from a matched font of an active font group or are synthesized.

cyan / cyan+	c / C
red / red+	r / R
magenta / magenta+	m/M
brown / brown+ (yellow)	b/B
white / white+	w / W

PLOTTING MODES	Escape Code
erase	-
inverse	!
noplot	_
rewrite	=
write	+
xor	*

FONTS	Escape Code
font number 1 - 9	f 1 - 9
font number 10 - 35	f a - z
font standard / text start	f @ / &

FONT CONTROL	Escape Code
bold on / off	F b / B
italics on / off	F i / I
shadow on / off	F s / S
shadow color	F / color
underline on / off	F u / U
underline color	F _ color

FUNCTIONS	Escape Code
size 1 - 9	1 - 9
spacing variable / fixed	v / V
narrow on / off	n / N
status default / text start	@ / &
erase color	d color
magin wrap / release	> / <
margin advance / set left	CR / l

CHARACTER FUNCTIONS	Escape Code
non-breaking space	#
non-breaking hyphen	h
new line	\
continued write	|

The **size, bold, italic** and **narrow** attributes either synthesize their effects starting with an individual font or select the best font match from a font group. See the font command and the

shadow	controls drop shadowing	5
underline	controls underlining	6
delay	sets delay time between characters	7
increment	adds extra spacing between characters	7
rotate	rotates text baseline	7
direction	controls plotting direction relative to baseline	8
margin	controls text wraparound at margin	9
align	controls alignment between fonts	10
reveal	displays rather than functions uncover text codes	12
measure	initializes text extent measurements	12
info	returns text attribute information	13

Description

Alters the way text appears on the screen. Text attributes are initialized to system defaults by the **initial**, **screen** or the **status restore; standard** commands. After selecting text attributes for a particular lesson, the modified settings can be saved and later restored by the **status** command. If the modified settings are saved in the **default** status buffer (say, by programming in the **+initial** control block), they will become the starting text attributes for any unit branched to by a **jump**.

While entering a line of text in the Source Editor, most of the **text** attributes can be manipulated directly in the text string through using uncover escape codes or the embedded command syntax. Following are the escape codes:

COLORS	Escape Code
black / black+ (gray)	a / A
blue / blue+	u / U
green / green+	g / G

```
status   restore; mystat; delete
```

status delete; 'NAME' | local

Deletes a saved status block from the memory pool.

Example

Deletes the *page1* set of saved settings from the memory pool.

```
status   delete; 'page1'
```

status reset

Deletes all named status blocks from the memory pool. The **default, local, arrow,** and **standard** buffers are unaffected.

System Variable

zreturn

-1	Operation successful
6	Duplicate name
10	Name does not exist (no previous status save)
16	Invalid name
18	Insufficient memory in memory pool

text

Controls text attributes.

text	keyword...		
	size	selects sized fonts	2
	bold	selects bold fonts	3
	italic	selects italic fonts	4
	narrow	selects narrow fonts	4
	spacing	controls character spacing	4

status restore; 'NAME' | local [; delete]

status restore; default | arrow

status restore; standard

Replaces the current plotting status with a previously saved set. If the palette was originally saved, it is restored. Optionally, the named or **local** block can be deleted from the memory pool by using the **delete** modifier

The **arrow** option restores the status in effect when the last **arrow** command was encountered. The palette is not restored.

The **standard** option restores the system standard plotting status, excluding the palette. To restore the standard palette use **palette init.**

Example 1

Restores the default plotting parameters (those last saved by a **status save;default** statement).

```
status   restore; default
```

Example 2

Saves and restores the plotting and palette settings over a library call. The library routine can alter the display settings as desired without affecting the calling program upon return. Alternately, the **status save** and **restore** could be built into the library routine to provide a more easily used tool.

```
status   save; local; palette
library  routines,graph
status   restore; local
```

Example 3

Restores the settings whose name is contained in the variable *mystat.* The memory block is then deleted.

The memory pool is used by the commands: **memory, image, window, status, area, flow, font and perm**. Memory blocks are tagged as belonging to a specific command type at creation and cannot be accessed by other commands using the memory pool; different commands can use the same name for a memory block without conflict.

Example 1

Plotting parameters are initialized in this **+initial** control block then saved in the default status buffer. These settings become the defaults for all subsequent main units that are reached by a **jump** branch. The **erase** command is used to bring the screen to the blue background even for the first branch into this lesson.

```
*                              in control block +initial
initial
color    yellow
colore   blue
size     2
font     geneva
status   save; default
erase
```

Example 2

Saves the current plotting parameters and palette in the memory pool under the name *status1*.

```
status   save; 'status1'; palette
```

BNSDOCID: <WO___9907007A2_I_>

bold; off

delay; 0

direction; right

increment; 0,0

italic; off

margin; wrap

narrow; off

reveal; off

rotate; 0

shadow; off,white

size; 1

spacing; fixed

underline; off,foregnd

- thick off

status save; 'NAME' | local [; palette]

status save; default [; palette]

Saves the current plotting status in a memory pool block or the **default** buffer. The name can be either a text literal or contained in a variable. Named blocks can be restored later in any unit The **palette** modifier causes the current palette settings to also be saved.

The **local** keyword saves the status in a memory pool block specific to the current unit. A local block can be restored only in the unit which saved it; it is deleted automatically when execution of the unit ends.

Saving the status to the **default** buffer makes it the default for all new main units entered by a **jump** branch.

maintain the plotting parameters over a subroutine call, or to provide a common look to all main units of a lesson by resetting the defaults. Restore options allow for the reinstatement of the system **standard** settings or the settings at the previous **arrow** in a unit.

status saves and restores these items:

- current screen location
- margins (both left and right)
- all loaded fonts, including current and standard fonts
- palette (if requested)

The settings of the following commands are also saved and restored by **status**. They are shown with their system **standard** settings that can be set to as a block with the **status restore;standard** option (or with the **initial** command).

- blink off
- color white
- colore black
- colorg black
- disable cursor
- enable font
- font standard; standard
- mode write
- origin 0,0
- rotate 0
- scale 1,1
- text align; baseline

After a **showv**, the current screen location is updated to the character position following the last character displayed.

Example

The **showa** plots "ABC" while the **showv** plots "AB C".

```
define   local
var,4
define   end
*
calc     var <=   h41   42    00    43    $$ A B null C
at       10:10
showa    var                              $$ ABC
at       11:10
showv    var                              $$ ABC
```

status

Saves and restores display status information.

---

status keyword...

save	saves current display settings in memory block
restore	restores display settings from memory block
delete	deletes named status memory blocks
reset	deletes all status memory blocks

Description

Saves the state of the current plotting parameters (such as the foreground and erase colors, mode, font, etc.) in a memory pool block or the default buffer. The saved status can later be restored to bring all plotting parameters back to their previous state. **status** can be used to

```
write   Anderson
        Carmen
        Engdahl
        Lillith
at      8:18
write   «t,rate(1),5,2»
        «t,rate(2),5,2»
        «t,rate(3),5,2»
        «t,rate(4),5,2»
at      8:31
write   $«t,rate(1)*2080,8,2»
        $«t,rate(2)*2080,8,2»
        $«t,rate(3)*2080,8,2»
        $«t,rate(4)*2080,8,2»
```

showv

Displays text buffer including null codes.

showv	buffer [,length]
write	« **showv** | v , buffer [,length] »

Description

Displays the contents of a buffer as alphanumeric characters, including any null characters (hex value h00) which are converted to space codes if they are actually plotted on the screen. Unlike **showv**, the **showa** command ignores null characters.

showv is primarily used in its embedded form in the tag of **pack** and **put**. In this case, **showv** causes null characters to be treated like any other character and does not ignore them or convert them to spaces.

Default Values

If the optional tags are omitted, the following default values are used:

IF the variable is...	Then default *field* and *right* are...
1-4 byte integer	8,0
8 byte integers and reals	12,3

showt displays special symbols to indicate the following error conditions:

****** The number to display does not fit in the specified field.

?????? The number is invalid. This is usually the result of an illegal floating point operation.

Example

Displays some entries in a table.

```
define   local
rate(4), 8, real          $$ pay rates end
define   end
*
calc     rate(1) <= 4.05
         rate(2) <= 12.76
         rate(3) <= 8.63
         rate(4) <= 5.25
at       5:3
write    Your payroll situation now looks like this:
at       7:3
write    EMPLOYEE        HOURLY RATE YEARLY SALARY
at
8:30
```

Example

Displays the exact hexadecimal contents of a real variable.

```
define   local
float, 8,   real        $$ a real variable
define   end
*
calc     float ⇐ π      $$ the value of pi
at       5:3
write    π looks like this in "hex".
at       6:3
showh    float
```

showt

Displays a numeric variable or expression in tabular format.

showt	variable [,field [, right]]
write	« **showt** | **t,** variable [,field [,right]] »
field	total size of display field including leading spaces, digits, and any decimal point
right	number of digits to right of decimal point

Description

Displays the value of a numeric variable or expression in a tabular right-justified format by inserting leading spaces and trailing zeroes as needed to fill the desired field.

After **showt**, the current screen location is updated to the character position following the last character displayed.

BNSDOCID: <WO___9907007A2_I_>

```
long      20
arrow     4:3
storea    stuname, zcount
ok
endarrow
at        8:3
write     Thank you, $$$ keep the trailing space
showa     stuname, zcount
write     .  $$ put the period on the end
```

Example 2

Displays the name of the current lesson and unit in the upper left comer:

```
at        1:1
write     « a,zlesson »:« a,zunit »
```

showh

Displays data in hexadecimal notation.

showh	buffer [,length]
write	« **showh** | h , buffer [,length] »

Description

Displays data in hexadecimal form for *length* digits. If *length* is omitted, enough digits are used to display the entire defined length of the variable. Each byte of a variable shows as two hexadecimal digits. Any length variable can be displayed.

After a **showh**, the current screen location is updated to the character position following the last character displayed.

```
1
*
at        1:65
write     Page      « show,page » of 50.
```

showa

Displays text stored in a variable.

showa	buffer [,length]
write	«**showa** | **a**, buffer [length]»

Description

Displays the contents of a variable buffer as alphanumeric characters for *length* characters. If *length* is omitted, the defined length of the buffer is used. The contents of any numeric variable can be shown, even if it contains non-numeric information.

After a **showa,** the current screen location is updated to the character position following the last character displayed. Any changes to plotting parameters resulting from escape code sequences in the displayed text remain in effect after the **showa**.

Examples

Example 1

The user's name is stored in *stuname* and then displayed using **showa**:

```
define    local
stuname, 20 $$ user name
define    end
*
at        3:3
write     Please enter your name.
```

updated to the character position following the last character displayed (without leading blanks or trailing zeroes).

The number of digits to the left of the decimal point is the value *of field* minus the value of *right* minus 1

show displays special symbols to indicate the following error conditions:

****** The number to display does not fit in the specified field.

?????? The number is invalid. This is usually the result of an illegal floating point operation.

If the optional tags are omitted the following default values are used:

IF the variable is	**Then default *field* and *right* are.**
1-4 byte integer	20,0
8 byte interger and reals	24,3

Examples

Example 1

Displays 10 quiz scores.

```
at        3:5
write     Here are your last 10 quiz scores:
loop      index <= 1, 10
.         at index+5:10
.         show     quizscor(index)
endloop
```

Example 2

Through use of an embedded **show** command in a write statement, a pagination title is put at the top of the display.

```
.       calc    array(2)    ⇐ 2
.       calc    array(3)    ⇐ 3
.       calc    array(4)    ⇐ 4
elseif  flag    = 0
.       calc    array(1)    ⇐ 4
.       calc    array(2)    ⇐ 3
.       calc    array(3)    ⇐ 2
.       calc    array(4)    ⇐ 1
elseif  flag    = 1             $$ skipped selection
elseif  flag    ≥ 2
.       calc    array(1)    ⇐ 1
.       calc    array(2)    ⇐ 2
.       calc    array(4)    ⇐ 8
endif
```

Note that in the last list assignment, array(3) is skipped.

show

Displays variables or numeric expressions.

show	expression [,field [,right]]
write	«**show** | s , expression [,field [,right]]»
field	maximum number of digits to display, including any decimal point
right	number of digits to display to the right of the decimal point

Description

Displays the value of a numeric variable or expression. Optional tags control the format of the numeric display. Values are displayed left-justified in the field and the screen location is

setc

Selectively sets variables to a list of values.

---

setc *SELECTOR ;* variable ⇐ value/s; value/s; ...

variable starting variable to assign list of values

Description

The selective form of **set**. Based on the value of the selector, **setc** selects one of the lists of values in the tag and assigns those values to consecutive variables.

Each list of values must be separated with a semicolon (;). Values within a list must be separated with a comma (,). A null value is used to skip an assignment. The tag of **setc** may be continued in the tag field of subsequent lines.

The starting variable need not be an array; consecutive following bytes of variable space are assigned based on the length and the type of the initial variable.

Example

Selectively assign a series of values to an array based on *flag*.

```
define   local
array(10), 4
flag, 1
define   end
*
setc     flag, array(1) ⇐ 1,2,3,4;
         4,3,2,1;;1,2,,8
```

The **setc** command above is equivalent to the following commands:

```
if       flag < 0
.        calc     array(1)    ⇐ 1
```

The arithmetic function **bit** (value,bit) is used to "read" set bits while **bitcnt** (value,bits) gives the count of bits set in a variable.

Example

One-bit flags in the 4-byte variable *correct* are used to keep track of the judgement for the 25 questions given a student. *index* contains the current question number. The questions are in the array *prob()* and the answers are in the array *answ()*. After the exam is completed, the **bitcnt** function is used to show the number of correct answers.

```
*       loop through all 25 questions, show the question, get the
*       answer, and set the flag appropriately
loop    index   <=      1, 25
.       at      5:3
.       showa           prob(index)       $$ display the question
.       arrow           20:3
.       answer          «showa, answ(index)»
.       .       setbit            correct, index, 1 $$ ok
.       no
.       . setbit            correct, index, 0       $$ wrong
.       . judge quit                    $$ exit the arrow
.       endarrow
.       erase           $$ clear screen for next question
endloop
*
at      25:3
write   The number correct is « s,bitcnt(correct,25) ».
```

```
define   local
array(4), 2
define   end
set      array(1) <= 15, 96, , score*100
         or
set      array(1) <= 15,      $$ continued across multiple lines
         96, ,
         score*100
```

The set commands above are equivalent to the following **calc** commands:

```
calc     array(1) <= 15
calc     array(2) <= 96
calc     array(4) <= score*100
```

setbit

Sets a bit in a variable.

setbit	variable,bit,value	
	variable	variable containing the bit to set
	bit	bit number to set
	value	value for bit (0 or 1)

Description

Sets a single bit in a variable. With this command, a single variable can be used for a number of one-bit flags, making more efficient use of variable space.

Bit 1 is the left-most (highest-order) bit. A bit number can be specified that crosses variable boundaries. If the value to set the bit to is other than zero or one, the right-most bit of the value is used.

```
...
seed      save            $$ save the seed
loop      index <= 1, 100
.         dot     index, randi(100)
endloop
pause
mode      erase
seed      save; set       $$ restore the saved seed
loop      index <= 1, 100
```

set

Assigns a series of values to consecutive variables.

---

set variable <= value 1, value2, ...

variable starting variable for assignments

Description

Assigns a list of values to consecutive memory locations, starting at the location of the specified variable. The first value to the right of the assignment arrow is assigned to *variable*. After that, consecutive memory locations of the same size and type as *variable* are assigned the remaining values in the tag, regardless of how subsequent variables may be defined. The variable given in set is often the first element of an array, but it may be any variable that is valid with **calc**. The tags of **set** may be continued in the tag field of subsequent lines.

A null "value" can be used to skip an assignment.

Example

Assigns a series of values to an array. Note that *array(3)* is not assigned and retains its original value.

zmaxpage Maximum number of pages available

seed

Obtains or sets the random number seed.

---

seed variable4 [; set]

Description

Used in the calculation of random numbers. Each time a new random number is generated, the system sets the seed to a new value. The random number seed is used by the **perm** command and the **randi** and **randf** functions

The current seed is obtained by using a 4-byte integer variable as the single argument to the **seed** command. The seed is set by specifying a 4-byte integer value (0 to 4,294,967,295) and using the **set** keyword. The system initializes the seed to the arbitrary value of **zclock** at start-up.

The **seed** command is normally used for repeatable random number sequences: the same random number sequence can be generated again by saving the initial seed and restoring it later. For example, if two permutation lists are generated starting from the same seed value, the lists will be identical.

Example

A scatter-graph of 100 random dots is first placed on the screen then erased after a pause. The seed is saved before the first generation of random dots then restored before the second "erase" generation of the same identical dots.

```
define   local
save,4,integer
index,2
define   end
```

2	80 x 25; text; mono	cga,text,high,mono
3	80 x 25; text; 16 color	cga,text,high,color
4	320 x 200; graphics; 4 color	cga,graphics,medium,color
5	320 x 200; graphics; mono	cga,graphics,medium,mono
6	640 x 200; graphics; mono	cga,graphics,high
7	80 x 25; text (on MDA)	cga,text
13	320 x 200; graphics; 16 color	ega,graphics,low
14	640 x 200; graphics; 16 color	ega,graphics,medium
16	640 x 350; graphics; 16 color	ega,graphics,high
17	640 x 480; graphics; 2 color	mcga,graphics,high
18	640 x 480; graphics; 16 color	vga,graphics,medium
19	320 x 200; graphics; 256 color	mcga,graphics,medium
106*	800 x 600; graphics; 16 color	vga,graphics,high
107*	800 x 600; graphics; 256 color	evga,graphics,high
108*	1024x 768; graphics; 16 color	vga,graphics.altl
109*	1024x 768; graphics; 256 color	evga.graphics,altl
113*	1280x1024; graphics; 16 color	vga,graphics,alt2
114*	1280x1024; graphics; 256 color	evga,graphics,alt2
*VESA-compatible display adapters only		

zscreenh Display driver file *(.DIS* file) in use (adjusted for the display hardware actually detected: e.g., if **EVGA.DIS** detected VGA hardware, **zscreenh** will be 10)

0 cga

2 Hercules

3 ega

9 att

10 vga

11 mcga

14 evga

zinaxcol Maximum number of colors available for current screen type

zpalincr Minimum change in RGB value required to have an effect on displayed color (see **zintincr**)

zintincr Minimum change in intensity value required to have an effect on displayed color (see **zpalincr**)

	zscreent	Screen type selected
	0	cga
	2	hercules
	3	ega
	9	att
	10	vga
	11	mcga
	14	evga
	zscreenm	Screen addressing mode
	0	graphics
	1	text
	zscreenr	Screen resolution
	0	low
	1	medium
	2	high
	3	alti
	4	alt2
	S	alt3
	zscreenc	Screen chrome
	0	color
	1	mono
	zscreen	Screen type as reported by BIOS

Value	Meaning	Screen Command
0	40 x 25; text; mono	cga,text,medium, mono
1	40 x 25; text; 16 color	cga,text,medium,color

SNSDOC'D <WO 9907007A2 I >

mcga, high, color	360x480	256	262,144	30:45	8x16	VS
vga, medium	640x480	16	262,144	30 80	8x16	VS
Enhanced VGA***						
vga, high	800x600	16	262,144	33 88	9x18	S
vga, alt1	1024x768	16	262,144	38 102	10x20	S
vga, alt2	1280x1024	16	262,144	42:106	12x24	S
evga, low	640x400	256	262,144	25:80	8x16	S
evga, medium	640x480	256	262,144	30:80	8x16	S
evga, high	800x600	256	262,144	33:88	9x18	S
evga, alt1	1024x768	256	262,144	38:102	10x20	S
evga, alt2	1280x1024	256	262,144	42:106	12x24	S

* when using *VGA.DIS* or *EVGA.DIS* the number of possible colors is 262,144

** display drivers supporting each screen are denoted by the following letters:

C CGA.DIS

E EGA.DIS

V VGA.DIS

S EVGA.DIS

***most Super VGA cards supported, either directly or through VESA standard

System Variables

zreturn

The **screen** command sets **zreturn** to the following values.

-2 ok, screen not changed; the requested screen already in effect

-1 ok, screen changed (not changed for **screen test** form)

0 Invalid screen for display driver file

1 Display adapter does not support requested screen

When a **screen** command executes successfully, the following set of system variables are reset They are not altered for the **screen test** form.

ENSDOCID: <WO__9907007A2 I >

Examples

Example 1

Selects the 640 x 480 x 16 color (vga,medium.graphics,color) screen. if the screen is not possible, a message is given before a branch exits the user back to the calling system.

```
screen vga
if      not(zreturn)   $$ see if can't set screen
.       write    Warning.
                 VGA Screen Type not available.
        pause
.       jump     =exit $$ exit to router or DOS
endif
```

Example 2

Tests whether the 1280 x 1024 x 256 color screen is available. If so, the high resolution version of a chemistry lesson is used. If not, the lower resolution version of the lesson is used.

```
screen   test, evga, alt2
if       zreturn $$ see if true(-1)
.        jump     chemHigh,start    $$ use high resolution version
else
.        jump     chemLow,start     $$ use lower resolution version
endif
```

Screen Types and Display Drivers

Screen	Pixels	Palette	Colors	Chars	CharSize	DisplayDrivers
IBM Standard						
cga, medium	320x200	4	16	20:40	8x10	CEVS
cga, high	640x200	2	16	20:80	8x10	CEVS
ega, low	320x200	16	16*	20:40	8x10	EVS
ega, medium	640x200	16	16*	20:80	8x10	EVS
ega, high	640x350	16	64*	25:80	8x14	EVS
mcga, medium	320x200	256	262,144	20:40	8x10	VS
mcga, high	640x480	2	262,144	30:80	8x16	VS

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 13/00 | A3

(11) International Publication Number: WO 99/07007

(43) International Publication Date: 11 February 1999 (11.02.99)

(21) International Application Number: PCT/US98/15627

(22) International Filing Date: 28 July 1998 (28.07.98)

(30) Priority Data:
08/902,591 29 July 1997 (29.07.97) US

(71) Applicant: CATHARON PRODUCTIONS, INC. [US/US]; 2119 Route 66, Ghent, NY 12075-2408 (US).

(72) Inventors: FEINBERG, Michael, A.; 2119 Route 66, Ghent, NY 12075-2408 (US). FEINBERG, Matthew, A.; 2119 Route 66, Ghent, NY 12075-2408 (US).

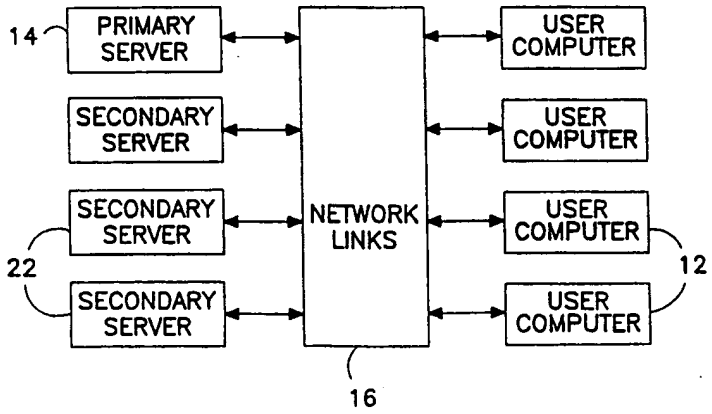
(74) Agent: SUDOL, R., Neil; Coleman Sudol, LLP, 13th floor, 270 Madison Avenue, New York, NY 10016 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, HR, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(88) Date of publication of the international search report:
14 May 1999 (14.05.99)

(54) Title: COMPUTERIZED SYSTEM AND ASSOCIATED METHOD FOR OPTIMALLY CONTROLLING STORAGE AND TRANSFER OF COMPUTER PROGRAMS ON A COMPUTER NETWORK



(57) Abstract

A computerized system and method for optimally controlling storage and transfer of computer programs between computers on a network to facilitate interactive program usage. In accordance with the method, an application program is stored in a nonvolatile memory (20) of a first computer (14) as a plurality of individual and independent machine-executable code modules. In response to a request from a second computer (12) transmitted over a network link, the first computer (14) retrieves a selected one of the machine-executable code modules and only that selected code module from the memory and transmits the selected code module over the network link to the second computer (12).

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

ENSDOCID: <WO 9907082 I >

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US98/15627

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :G06F 13/00
US CL :395/200.3,200.33,200.46,200.62,200.48,200.49,675

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/200.3,200.33,200.46,200.62,200.48,200.49,675

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 5,475,819 A (MILLER et al.) 12 December 1995, col. 2, lines 5-43, col. 3, lines 5-60, col. 5 line 16 to col. 7, line 48.	1-2,4-8, 10-17, 21-2 7, 29-32, 34-36
Y	US 5,341,477 A (PITKIN et al.) 23 August 1994, col. 2, line 35 to col. 8, line 37.	1-2, 8, 16-17, 26-27, 30-32,36
Y,P	US 5,774,660 A (BRENDEL et al.) 30 June 1998, col. 10 line 38 to col. 12, line 55.	1-2,4-8, 10-17, 21-2 7, 29-32, 34-36
Y,P	US 5,727,950 A (COOK et al.) 17 March 1998, col. 20, line 49 to	1-2,4-8

[X] Further documents are listed in the continuation of Box C. [ ] See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 22 JANUARY 1999	Date of mailing of the international search report 19 MAR 1999
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 305-3230	Authorized officer ZARNI MAUNG Telephone No. (703) 305-3900

Form PCT/ISA/210 (second sheet)(July 1992)*

BNSDOCID: <WO__ 9907007A3_I_>

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US98/15627

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 5,819,020 A (BEELER, Jr.) 06 October 1998, see the whole reference.	1-36
A	US 5,828,847 A (GEHR et al.) 27 October 1998, see the whole reference.	1-36

Form PCT/ISA/210 (continuation of second sheet)(July 1992)*